A Tao of Regular Expressions

Steve Mansour
sman@scruznet.com
Revised: June 5, 1999
(copied by jm /at/ jmason.org from http://www.scruz.net/%7esman/regexp.htm, after the original disappeared!)

C O N T E N T S

What Are Regular Expressions

A regular expression is a formula for matching strings that follow some pattern. Many people are afraid to use them because they can look confusing and complicated. Unfortunately, nothing in this write up can change that. However, I have found that with a bit of practice, it's pretty easy to write these complicated expressions. Plus, once you get the hang of them, you can reduce hours of laborious and error-prone text editing down to minutes or seconds. Regular expressions are supported by many text editors, class libraries such as Rogue Wave's Tools.h++, scripting tools such as awk, grep, sed, and increasingly in interactive development environments such as Microsoft's Visual C++.

Regular expressions usage is explained by examples in the sections that follow. Most examples are presented as **`vi`** substitution commands or as grep file search commands, but they are representative examples and the concepts can be applied in the use of tools such as sed, awk, perl and other programs that support regular expressions. Have a look at Regular Expressions In Various Tools for examples of regular expression usage in other tools. A short explanation of vi's substitution command and syntax is provided at the end of this document.

Regular Expression Basics

Regular expressions are made up of normal characters and *metacharacters*. Normal characters include upper and lower case letters and digits. The metacharacters have special meanings and are described in detail below.

In the simplest case, a regular expression looks like a standard search string. For example, the regular expression "testing" contains no metacharacters. It will match "testing" and "123testing" but it will not match "Testing".

To really make good use of regular expressions it is critical to understand metacharacters. The table below lists metacharacters and a short explanation of their meaning.

Metacharacter	*Description*
.	Matches any single character. For example the regular expression **r.t** would match the strings *rat*, *rut*, *r t*, but not *root*.
$	Matches the end of a line. For example, the regular expression **weasel$** would match the end of the string "*He's a weasel*" but not the string "*They are a bunch of weasels.*"
^	Matches the beginning of a line. For example, the regular expression **^When in** would match the beginning of the string "*When in the course of human events*" but would not match "*What and When in the*" .
*****	Matches zero or more occurences of the character immediately preceding. For example, the regular expression **.*** means match any number of any characters.
****	This is the quoting character, use it to treat the following character as an ordinary character. For example, **\$** is used to match the dollar sign character ($) rather than the end of a line. Similarly, the expression \. is used to match the period character rather than any single character.
[ ] **[c1-c2]** **[^c1-c2]**	Matches any one of the characters between the brackets. For example, the regular expression **r[aou]t** matches *rat*, *rot*, and *rut,* but not *ret*. Ranges of characters can specified by using a hyphen. For example, the regular expression **[0-9]** means match any digit. Multiple ranges can be specified as well. The regular expression **[A-Za-z]** means match any upper or lower case letter. To match any character *except* those in the range, the complement range, use the caret as the first character after the opening bracket. For example, the expression **[^269A-Z]** will match any characters except 2, 6, 9, and upper case letters.
\< \>	Matches the beginning (\<) or end (\>) or a word. For example, **\<the** matches on "the" in the string "*for the wise*" but does not match "the" in "*otherwise*". NOTE: this metacharacter is not supported by all applications.
\(\)	Treat the expression between \(and \) as a group. Also, saves the characters matched by the expression into temporary holding areas. Up to nine pattern matches can be saved in a single regular expression. They can be referenced as **\1** through **\9**.
|	Or two conditions together. For example **(him|her)** matches the line "*it belongs to him*" and matches the line "*it belongs to her*" but does not match the line "*it belongs to them.*" NOTE: this metacharacter is not supported by all applications.
+	Matches one or more occurences of the character or regular expression immediately preceding. For example, the regular expression **9+** matches 9, 99, 999. NOTE: this metacharacter is not supported by all applications.
?	Matches 0 or 1 occurence of the character or regular expression immediately preceding.NOTE: this metacharacter is not supported by all applications.
\{*i*\} **\{*i*,*j*\}**	Match a specific number of instances or instances within a range of the preceding character. For example, the expression **A[0-9]\{3\}** will match "A" followed by exactly 3 digits. That is, it will match A123 but not A1234. The expression **[0-9]\{4,6\}** any sequence of 4, 5, or 6 digits. NOTE: this metacharacter is not supported by all applications.

The simplest metacharacter is the dot. It matches any one character (excluding the newline character). Consider a file named test.txt consisting of the following lines:

```
he is a rat
```

```
he is in a rut
the food is Rotten
I like root beer
```

We can use grep to test our regular expressions. Grep uses the regular expression we supply and tries to match it to every line of the file. It prints all lines where the regular expression matches at least one sequence of characters on a line. The command

```
grep r.t test.txt
```

searches for the regular expression **`r.t`** in each line of test.txt and prints the matching lines. The regular expression **`r.t`** matches an **`r`** followed by any character followed by a **`t`**. It will match **`rat`** and **`rut`**. It does not match the **`Rot`** in **`Rotten`** because regular expressions are case sensitive. To match both the upper and lower the square brackets (character range metacharacters) can be used. The regular expression **`[Rr]`** matches either **`R`** or **`r`**. So, to match an upper or lower case **`r`** followed by any character followed by the character **`t`** the regular expression **`[Rr].t`** will do the trick.

To match characters at the beginning of a line use the circumflex character (sometimes called a caret). For example, to find the lines containing the word "he" at the beginning of each line in the file test.txt you might first think the use the simple expression **`he`**. However, this would match **`the`** in the third line. The regular expression **`^he`** only matches the **`h`** at the beginning of a line.

Sometimes it is easier to indicate something what should not be matched rather than all the cases that should be matched. When the circumflex is the first character between the square brackets it means to match any character which is not in the range. For example, to match **`he`** when it is not preceded by **`t`** or **`s`**, the following regular expression can be used: **`[^st]he`**.

Several character ranges can be specified between the square brackets. For example, the regular expression **`[A-Za-z]`** matches any letter in the alphabet, upper or lower case. The regular expression **`[A-Za-z][A-Za-z]*`** matches a letter followed by zero or more letters. We can use the **`+`** metacharacter to do the same thing. That is, the regular expression **`[A-Za-z]+`** means the same thing as **`[A-Za-z][A-Za-z]*`**. Note that the **`+`** metacharacter is not supported by all programs that have regular expressions. See Regular Expressions Syntax Support for more details.

To specify the number of occurrences matched, use the braces (they must be escaped with a backslash). As an example, to match all instances of **`100`** and **`1000`** but not **`10`** or **`10000`** use the following: **`10\{2,3\}`**. This regular expression matches a the digit **`1`** followed by either 2 or 3 **`0`**'s. A useful variation is to omit the second number. For example, the regular expression **`0\{3,\}`** will match 3 or more successive **`0`**'s.

Simple Examples

Here are a few representative, simple examples.

vi command	*What it does*
`:%s/ */ /g`	Change 1 or more spaces into a single space.
`:%s/ *$//`	Remove all spaces from the end of the line.

`:%s/^/ /`	Insert a space at the beginning of every line.
`:%s/^[0-9][0-9]* //`	Remove all numbers at the beginning of a line.
`:%s/b[aeio]g/bug/g`	Change all occurences of *bag*, *beg*, *big*, and *bog*, to *bug*.
`:%s/t\([aou]\)g/h\1t/g`	Change all occurences of *tag*, *tog*, and *tug* to *hat*, *hot*, and *hug* respectively.

Medium Examples (Strange Incantations)

Example 1

Change all instances of foo(*a,b,c*) to foo(*b,a,c*). where a, b, and c can be any parameters supplied to foo(). That is, we must be able to make changes like the following:

Before	After
`foo(10,7,2)`	`foo(7,10,2)`
`foo(x+13,y-2,10)`	`foo(y-2,x+13,10)`
`foo( bar(8), x+y+z, 5)`	`foo( x+y+z, bar(8), 5)`

The following substitution command will do the trick :

```
:%s/foo(\([^,]*\),\([^,]*\),\([^)]*\))/foo(\2,\1,\3)/g
```

Now, let's break this apart and analyze what's happening. The idea behind this expression is to identify invocations of foo() with three parameters between the parentheses. The first parameter is identified by the regular expression `\([^,]*\),` which we can analyze from the inside out.

`[^,]`	means any character which is not a comma
`[^,]*`	means 0 or more characters which are not commas
`\([^,]*\)`	tags the non-comma characters as `\1` for use in the replacement part of the command
`\([^,]*\),`	means that we must match 0 or more non-comma characters which are followed by a comma. The non-comma characters are tagged.

This is a good time to point out one of the most common problems people have with regular expressions. Why would we use an expression like `[^,]*,` instead of something more straightforward like `.*,` to match the first parameter? Consider applying the pattern `.*,` to the string "10,7,2". Should it match "10," or "10,7," ? To resolve this ambiguity, regular expressions will always match the longest string possible. In this case "10,7," which covers two parameters instead of one parameter like we want. So, by using the expression `[^,]*,` we force the pattern to match all characters up to the first comma.

The expression up to this point is: `foo(\([^,]*\),` and can be roughly translated as "after you find `foo(` tag all characters up to the next comma as `\1`". We tag the second parameter just like the first and it can be referenced as `\2`. The tag used on the third parameter is exactly like the others except that we search for all characters up to the right parenthesis. It may be superfluous to search for the last parameter since we don't have to move it. But this pattern guarantees that we update only those instances of foo() where 3 parameters are specified. In these times of function and method overloading, being explicit often proves to be useful. In the substitution portion of the command, we explicitly enter the invocation of

foo() as we want it, referencing the matched patterns in the new order where the first and second parameter have been switched.

Example 2

We have a CSV (comma separated value) file with information we need, but in the wrong format. The columns of data are currently arranged in the following order: Name, Company Name, State, Postal Code. We need to reorganize the data into the following order in order to use it with a particular piece of software: Name, State-Postal Code, Company Name. This means that we must change the order of the columns in addition to merging two columns to form a new column value. The particular piece of software that needs this data will not work if there are any whitespace characters (spaces or tabs) before or after the commas. So we must remove whitespace around the commas.

Here are a few lines from the data we have:

```
Bill Jones,     HI-TEK Corporation ,  CA, 95011
Sharon Lee Smith,  Design Works Incorporated,  CA, 95012
B. Amos   ,  Hill Street Cafe,  CA, 95013
Alexander Weatherworth,  The Crafts Store,  CA, 95014
...
```

We need to transform them to look like this:

```
Bill Jones,CA 95011,HI-TEK Corporation
Sharon Lee Smith,CA 95012,Design Works Incorporated
B. Amos,CA 95013,Hill Street Cafe
Alexander Weatherworth,CA 95014,The Crafts Store
...
```

We'll look at two regular expressions to solve this problem. The first moves the columns around and merges the data. The second removes the excess spaces.

Here is the first pass at a substitution command that will solve the problem:

```
:%s/\([^,]*\),\([^,]*\),\([^,]*\),\(.*\)/\1,\3 \4,\2/
```

The approach is similar to that of Example 1. The Name is matched by the expression `\([^,]*\),` that is, all characters up to the first comma. The name can then be referenced as `\1` in the replacement pattern. The Company Name and State fields are matched just like the Name field and are referenced as `\2` and `\3` in the replacement pattern. The last field is matched with the expression `\(.*\)` which can be translated as "match all characters through the end of the line". The replacement pattern is constructed by calling out each tagged expression in the appropriate order and adding or not adding the delimeter.

The following substitution command will remove the excess spaces:

```
:%s/[ \t]*,[ \t]*/,/g
```

To break it down: `[ \t]` matches a space or tab character; `[ \t]*` matches 0 or more spaces or tabs; `[ \t]*`, matches 0 or more spaces or tabs followed by a comma; and finally `[ \t]*,[ \t]*` matches 0 or more spaces or tabs followed by a comma followed by 0 or more spaces or tabs. In the replacement pattern, we simply replace whatever we matched with a single comma. The optional `g` parameter is added to the end of the substitution command to apply the substitution to all commas in the line.

Example 3

Suppose you have a multi-character sequence that repeats. For example, consider the following:

```
Billy tried really hard
Sally tried really really hard
Timmy tried really really really hard
Johnny tried really really really really hard
```

Now suppose you want to change "really", "really really", and any number of consecutive "really" strings to a single word: "very". The command

```
:%s/\(really \)\(really \)*/very /
```

changes the text above to:

```
Billy tried very hard
Sally tried very hard
Timmy tried very hard
Johnny tried very hard
```

The expression **`\(really \)*`** matches 0 or more sequences of "really ". The sequence **`\(really \)\(really \)*`** matches one or more instances of the sequence "really ".

Hard Examples (Magical Hieroglyphics)

coming soon.

Regular Expressions In Various Tools

OK, you'd like to use regular expressions, but you can't bring yourself to use vi. Here, then, are a few examples of how to use regular expressions in other tools. Also, I have attempted to summarize the differences in regular expressions you will find between different programs.

You can use regular expressions in the Visual C++ editor. Select Edit->Replace, then be sure to check the checkbox labled "Regular expression". For vi expressions of the form **`:%s/pat1/pat2/g`** set the Find What field to **`pat1`** and the Replace with field to **`pat2`**. To simulate the range (% in this case) and the **`g`** option you will have to use the Replace All button or appropriate combinations of Find Next and Replace

sed

Sed is a **S**tream **ED**itor which can be used to make changes to files or pipes. For complete details, see the man page sed(1).

Here are a few interesting sed scripts. Assume that we're processing a file called price.txt. Note that the edits don't actually happen to the input file, sed simply processes each line of the file with the command you supply and echos the result to its standard out.

sed script	*Description*
`sed 's/^$/d' price.txt`	removes all empty lines
`sed 's/^[ \t]*$/d' price.txt`	removes all lines containing only whitespace
`sed 's/"//g' price.txt`	remove all quotation marks

awk

Awk is a programming language which can be used to perform sophisticated analysis and manipulation of text data. For complete details, see the man page awk(1). Its peculiar name is an acronym made up of the first character of its authors last names (Aho, Weinberger, and Kernighan).

There are many good awk examples in the book The AWK Programming Language (written by Aho, Weinberger, and Kernighan). Please don't form any broad opinions about awk's capabilities based on the following trivial sample scripts. For purposes of these examples, assume that we're working with a file called price.txt. As with sed, awk simply echos its output to its standard out.

awk script	*Description*
`awk '$0 !~ /^$/' price.txt`	removes all empty lines
`awk 'NF > 0' price.txt`	a better way to remove all lines in awk
`awk '$2 ~ /^[JT]/ {print $3}' price.txt`	print the third field of all lines whose second field begins with 'J' or 'T'
`awk '$2 !~ /[Mm]isc/ {print $3 + $4}' price.txt`	for all lines whose second field does not contain 'Misc' or 'misc' print the sum of columns 3 and 4 (assumed to be numbers).
`awk '$3 !~ /^[0-9]+\.[0-9]*$/ {print $0}' price.txt`	print all lines where field 3 is not a number. The number must be of the form: `d.d` or `d.` where `d` is any number of digits from 0 to 9.
`awk '$2 ~ /John|Fred/ {print $0}' price.txt`	print the entire line if the second field contains 'John' or 'Fred'

grep

grep is a program used to match regular expressions in one or more specified files or in an input stream. Its name programming language which can be used to perform data manipulation on files or pipes. For complete details, see the man page grep(1). Its peculiar name stems from its roots as a command in vi, **g/***re***/p** meaning **g**lobal **r**egular **e**xpression **p**rint.

For the examples below, assume we have the text below in a file named phone.txt. Its format is last name followed by a comma, first name followed by a tab, then a phone number.

```
Francis, John           5-3871
Wong, Fred              4-4123
Jones, Thomas           1-4122
Salazar, Richard        5-2522
```

grep command	*Description*
`grep '\t5-...1' phone.txt`	print all the lines in phone.txt where the phone number begins with 5 and ends with 1. Note that the tab character is represented by `\t`.
`grep '^S[^ ]* R' phone.txt`	print lines where the last name begins with S and first name begins with R.
`grep '^[JW]' phone.txt`	print lines where the last name begins with J or W
`grep ', ....\t' phone.txt`	print lines where the first name is 4 characters. The tab character is represented by `\t`.
`grep -v '^[JW]' phone.txt`	print lines that do not begin with J or W
`grep '^[M-Z]' phone.txt`	print lines where the last name begins with any letter from M to Z.
`grep '^[M-Z].*[12]' phone.txt`	print lines where the last name begins with a letter from M to Z and where the phone number ends with a 1 or 2.

egrep

egrep is an extended version of grep. It supports a few more metacharacters in its regular expressions. For the examples below, assume we have the text below in a file named phone.txt. Its format is last name followed by a comma, first name followed by a tab, then a phone number.

```
Francis, John           5-3871
Wong, Fred              4-4123
Jones, Thomas           1-4122
Salazar, Richard        5-2522
```

egrep command	*Description*
`egrep '(John|Fred)' phone.txt`	print all lines that contain the name *John* or *Fred*.
`egrep 'John|22$|^W' phone.txt`	print lines that contain *John* or that end with 22 or that begin with *W*.
`egrep 'net(work)?s' report.txt`	print lines in report.txt contain *networks* or *nets*.

Regular Expressions Syntax Support

Command or Environment	.	[ ]	^	$	\(\)	\{ \}	?	+	|	( )
vi	X	X	X	X	X					
Visual C++	X	X	X	X	X					
awk	X	X	X	X			X	X	X	X
sed	X	X	X	X	X	X				
Tcl	X	X	X	X	X		X	X	X	X

ex	X	X	X	X	X	X				
grep	X	X	X	X	X	X				
egrep	X	X	X	X	X		X	X	X	X
fgrep	X	X	X	X	X					
perl	X	X	X	X	X		X	X	X	X

The vi Substitution Command

Vi's substitution command has the form

> **`:`***range***`s/`***pat1***`/`***pat2***`/g`**

where

> **`:`** begins an ex (command line editor) command which is applied to the file currently being edited.
>
> *range* is the line range specifier. Use the percent sign (%) to indicate all lines. Use the dot (.) to indicate the current line. Use the dollar sign to indicate the last line. You can also use specific line numbers. Examples: **`10,20`** means lines 10 through 20; **`.,$`** means from the current line to the last line; **`.+2,$-5`** means from two lines after the current through the fifth line up from the end of the file.
>
> **`s`** is the substitution command.
>
> *pat1* is the regular expression to be searched for. This paper is full of examples.
>
> *pat2* is the replacement pattern. This paper is full of examples.
>
> **`g`** is optional. When present the substitution is made to all matches on the line. When it is not present, the substitution is applied only to the first match on the line.

There are many online manuals for vi that provide more complete detail. This page has a number of good vi links and information.

[Back to Home Page]