Contents

I. News
II. Introduction
- 2.1 What is VIM?
- 2.2 About this Tutorial
- 2.3 Credits

III. Substitute Command
- 3.1 Search & Replace
- 3.2 Line Ranges & Addressing

IV. Pattern Description
- 4.1 Anchors
- 4.2 "Escaped" characters or metacharacters
- 4.3 Quantifiers, Greedy and Non-Greedy
- 4.4 Character ranges
- 4.5 Grouping and Backreferences
- 4.6 Alternations
- 4.7 Operator Precedence

V. Global Command
- 5.1 Global search and execution
- 5.2 Examples

VI. Examples
- 6.1 Tips & Techniques
- 6.2 Creating Outline
- 6.3 Working with Tables

VII. Other Regexp Flavors
VIII. Links

I. News

- This page has been moved from an old geocities to rescue it from a premature death. If you are the former maintainer of this content please contact us (we tried to contact you, honest) to let us know if you are interested in resume maintainence of this content or just to say Cheers!

II. Introduction

2.1 What is VIM?

Vim is an improved (in many ways) version of vi, a ubiquitous text editor found on any UNIX system. VIM was created by Bram Moolenaar with a help of other people. It's free but if you like it you can make a charitable contribution to orphans in Uganda.

Vim has its own web site, **www.vim.org** and several mailing lists, with a wealth of information on every aspect of VIM. Vim was successfully ported to nearly all existing OS. It is a default editor in many Linux distributions (e.g. RedHat).

VIM has all features of a modern programmer's editor - macro language, syntax highlighting, customizable user interface, easy integration with various IDEs plus a set of features which makes VIM so attractive to its users: crash recovery, automatic commands, session management.

VIM has a very broad and loyal user base. Over 10 million people have it installed (counting only Linux users). Estimation is that there are about half a million people using Vim as their main editor. And this number is growing.

2.2 About this Tutorial

I started this tutorial for one simple reason - I like regular expressions. Nothing compares to the satisfaction from a well-crafted regexp which does exactly what you wanted it to do :-). I hope it's passable as a foreword.

Speaking more seriously, regular expressions (or regexps for short) are tools used to manipulate text and data. They don't exist as a standalone product but usually are a part of some program/utility. The best known example is UNIX *grep,* a program to search files for lines that match certain pattern. The search pattern is described in terms of *regular expressions.* You can think of regexps as a specialized pattern language. Regexps are quite useful and can greatly reduce time it takes to do some tedious text editing.

(Regexp terminology is largely borrowed from Jeffrey Friedl "Mastering Regular Expressions.")

2.3 Credits

Many thanks (in no particular order): Benji Fisher, Zdenek Sekera, Preben "Peppe" Guldberg, Steve Kirkendall, Shaul Karl and all others who helped me with their comments.

Feel free to send me (**volontir at yahoo dot com**) your comments. suggestions, examples...

III. Substitute Command

3.1 Search & Replace

So, what can you do with regular expressions? The most common task is to make replacements in a text following some certain rules. For this tutorial you need to know VIM search and replace command (S&R) `:substitute`. Here is an excerpt from VIM help:

:*range* s[ubstitute]/*pattern*/*string*/cgiI

For each line in ***the range*** replace a match of ***the pattern*** with ***the string*** where:

c	Confirm each substitution
g	Replace all occurrences in the line (without **g** - only first).
i	Ignore case for the pattern.
I	Don't ignore case for the pattern.

Part of the command word enclosed in the "[" & "]" can be omitted.

3.2 Range of Operation, Line Addressing and Marks

Before I begin with a pattern description let's talk about line addresses in Vim. Some Vim commands can accept a line range in front of them. By specifying the line range you restrict the command execution to this particular part of text only. Line range consists of one or more line specifiers, separated with a comma or semicolon. You can also mark your current position in the text typing `ml` , where *"l"* can be any letter, and use it later defining the line address.

Specifier	Description
number	an absolute line number
.	the current line
$	the last line in the file
%	the whole file. The same as **1,$**
'*t*	position of mark "t"
/*pattern*[/]	the next line where text *"pattern*" matches.
?*pattern*[?]	the previous line where text "*pattern*" matches
\/	the next line where the previously used search pattern matches
\?	the previous line where the previously used search pattern matches
\&	the next line where the previously used substitute pattern matches

If no line range is specified the command will operate on the current line only.

Here are a few examples:

```
10,20
```

- from 10 to 20 line.

Each may be followed (several times) by "+" or "-" and an optional number. This number is added or subtracted from the preceding line number. If the number is omitted, 1 is used.

```
/Section 1/+,/Section 2/-
```

- all lines between **Section 1** and **Section 2**, non-inclusively, i.e. the lines containing **Section 1** and **Section 2** will not be affected.

The `/pattern/` and `?pattern?` may be followed by another address separated by a semicolon. A semicolon between two search patterns tells Vim to find the location of the first pattern, then start searching from that location for the second pattern.

```
/Section 1/;/Subsection/-,/Subsection/+
```

- first find **Section 1**, then the first line with **Subsection**, step one line down (beginning of the range) and find the next line with **Subsection**, step one line up (end of the range).

The next example shows how you can reuse you search pattern:

```
:/Section/+ y
```

- this will search for the **Section** line and yank (copy) one line after into the memory.

```
:// normal p
```

- and that will search for the next **Section** line and put (paste) the saved text on the next line.

Tip 1: frequently you need to do S&R in a text which contains UNIX file paths - text strings with slashes ("/") inside. Because S&R command uses slashes for pattern/replacement separation you have to escape every slash in your pattern, i.e. use "\/" for every "/" in your pattern:

```
s/\/dir1\/dir2\/dir3\/file/dir4\/dir5\/file2/g
```

To avoid this so-called "backslashitis" you can use different separators in S&R (I prefer ":")

```
s:/dir1/dir2/dir3/file:/dir4/dir5/file2:g
```

Tip 2: You may find these mappings useful (put them in your **.vimrc** file)

```
noremap ;; :%s:::g<Left><Left><Left>
noremap ;' :%s:::cg<Left><Left><Left><Left>
```

These mappings save you some keystrokes and put you where you start typing your search pattern. After typing it you move to the replacement part , type it and hit return. The second version adds confirmation flag.

IV. Pattern Description

4.1 Anchors

Suppose you want to replace all occurrences of **vi** with **VIM**. This can be easily done with

```
s/vi/VIM/g
```

If you've tried this example then you, no doubt, noticed that **VIM** replaced all occurrences of **vi** even if it's a part of the word (e.g. na**vi**gator). If we want to be more specific and replace only whole words **vi** then we need to correct our pattern. We may rewrite it by putting spaces around **vi**:

```
s: vi : VIM :g
```

But it will still miss **vi** followed by the punctuation or at the end of the line/file. The right way is to put special word boundary symbols "`\<`" and "`\>`" around **vi**.

```
s:\<vi\>:VIM:g
```

The beginning and the end of the line have their own special anchors - "`^`" and "`$`", respectively. So, for all **vi** only at the start of the line:

```
s:^vi\>:VIM:
```

To match the lines where **vi** is the only word:

```
s:^vi$:VIM:
```

Now suppose you want to replace not only all **vi** but also **Vi** and **VI**. There are several ways to do this:

- probably the simplest way is to put "i" - ignore case in a pattern `%s:vi:VIM:gi`
- define a class of characters. This is a sequence of characters enclosed by square brackets "[" and "]". It matches any character from this set. So `:%s:[Vv]i:VIM:` will match **vi** and **Vi**. More on character ranges in the following section.

4.2 "Escaped" characters or metacharacters

So far our pattern strings were constructed from normal or ***literal*** text characters. The power of regexps is in the use of ***metacharacters***. These are types of characters which have special meaning inside the search pattern. With a few exceptions these metacharacters are distinguished by a "magic" backslash in front of them. The table below lists some common VIM metacharacters.

#	Matching	#	Matching
.	any character except new line		
\s	whitespace character	**\S**	non-whitespace character
\d	digit	**\D**	non-digit
\x	hex digit	**\X**	non-hex digit
\o	octal digit	**\O**	non-octal digit
\h	head of word character (a,b,c...z,A,B,C...Z and _)	**\H**	non-head of word character
\p	printable character	**\P**	like **\p**, but excluding digits
\w	word character	**\W**	non-word character
\a	alphabetic character	**\A**	non-alphabetic character
\l	lowercase character	**\L**	non-lowercase character

\u	uppercase character	**\U**	non-uppercase character

So, to match a date like 09/01/2000 you can use (assuming you don't use "/" as a separator in the S&R)

```
\d\d/\d\d/\d\d\d\d
```

To match 6 letter word starting with a capital letter

```
\u\w\w\w\w\w
```

Obviously, it is not very convenient to write `\w` for any character in the pattern - what if you don't know how many letters in your word? This can be helped by introducing so-called *quantifiers*.

4.3 Quantifiers, Greedy and Non-Greedy

Using quantifiers you can set how many times certain part of you pattern should repeat by putting the following after your pattern:

Quantifier	Description
*****	matches 0 or more of the preceding characters, ranges or metacharacters .* matches everything including empty line
\+	matches 1 or more of the preceding characters...
\=	matches 0 or 1 more of the preceding characters...
\{n,m}	matches from n to m of the preceding characters...
\{n}	matches exactly n times of the preceding characters...
\{,m}	matches at most m (from 0 to m) of the preceding characters...
\{n,}	matches at least n of of the preceding characters...
	where **n** and **m** are positive integers (>0)

Now it's much easier to define a pattern that matches a word of *any* length `\u\w\+`.

These quantifiers are *greedy* - that is your pattern will try to match ***as much text as*** possible. Sometimes it presents a problem. Let's consider a typical example - define a pattern to match delimited text, i.e. text enclosed in quotes, brackets, etc. Since we don't know what kind of text is inside the quotes we'll use

```
/".*"/
```

But this pattern will match *everything* between the first " and the last " in the following line:

```
this file is normally "$VIM/.gvimrc". You can check this with ":version".
```

This problem can be resolved by using non-greedy quantifiers:

Quantifier	Description
\{-}	matches 0 or more of the preceding atom, as few as possible
\{-n,m}	matches 1 or more of the preceding characters...
\{-n,}	matches at lease or more of the preceding characters...
\{-,m}	matches 1 or more of the preceding characters...
	where **n** and **m** are positive integers (>0)

Let's use `\{-}` in place of `*` in our pattern. So, now `".\{-}"` will match the first quoted text:

```
this file is normally "$VIM/gvimrc". You can check this with ":version".
```

`.\{-}` pattern is not without surprises. Look what will happen to the following text after we apply:

```
:s:.\{-}:_:g
```

Before:

```
n and m are decimal numbers between
```

After:

```
_n_ _a_n_d_ _m_ _a_r_e_ _d_e_c_i_m_a_l_ _n_u_m_b_e_r_s_ _b_e_t_w_e_e_n_
```

"As few as possible" applied here means zero character replacements. However match **does occur** between characters! To explain this behavior I quote Bram himself:

Matching zero characters is still a match. Thus it will replace zero characters with a "_". And then go on to the next position, where it will match again.

It's true that using "\{-}" is mostly useless. It works this way to be consistent with "", which also matches zero characters. There are more useless ones: "x\{-1,}" always matches one x. You could just use "x". More useful is something like "x\{70}". The others are just consistent behavior: ..., "x\{-3,}", "x\{-2,}", "x\{-1,}.*

- Bram

But what if we want to match only the second occurrence of quoted text? Or we want to replace only a part of the quoted text keeping the rest untouched? We will need *grouping* and *backreferences.* But before let's talk more about character ranges.

4.4 Character ranges

Typical character ranges:

`[012345]` will match any of the numbers inside the brackets. The same range can be written as `[0-5]`, where dash indicates a range of characters in ASCII order. Likewise, we can define the range for all lowercase letters: `[a-z]`, for all letters: `[a-zA-Z]`, letters and digits: `[0-9a-zA-Z]` etc. Depending on your system locale you can define range which will include characters like à, Ö, ß and other non ASCII characters.

Note that the range represents just *one character* in the search pattern, that is `[0123]` and `0123` are not the same. Likewise the order (with a few exceptions) is not important: `[3210]` and `[0123]` are the same character ranges, while `0123` and `3210` are two different patterns. Watch what happens when we apply

```
s:[65]:Dig:g
```

to the following text:

Before:

```
High 65 to 70. Southeast wind around 10
```

After:

```
High DigDig to 70. Southeast wind around 10
```

and now:

```
s:65:Dig:g
```

Before:

```
High 65 to 70. Southeast wind around 10
```

After:

```
High Dig to 70. Southeast wind around 10
```

Sometimes it's easier to define the characters you don't want to match. This is done by putting a negation sign `"^"` (caret) as a first character of the range

/`[^A-Z]`/

- will match *any character* except capital letters. We can now rewrite our pattern for quoted text using

`/"[^"]\+"`/

Note: inside the [ ] all metacharacters behave like ordinary characters. If you want to include "-" (dash) in your range put it first

```
/[-0-9]/
```

- will match all digits *and* -. "^" will lose its special meaning if it's not the first character in the range.

Now, let's have some real life example. Suppose you want to run a grammar check on your file and find all places where new sentence does not start with a capital letter. The pattern that will catch this:

```
\.\s\+[a-z]
```

- a period followed by one or more blanks and a lowercase word. We know how to find an error, now let's see how we can correct it. To do this we need some ways to remember our matched pattern and recall it later. That is exactly what *backreferences* are for.

4.5 Grouping and Backreferences

You can group parts of the pattern expression enclosing them with "`\(`" and "`\)`" and refer to them inside the replacement pattern by their special number `\1, \2 ... \9`. Typical example is swapping first two words of the line:

```
s:\(\w\+\)\(\s\+\)\(\w\+\):\3\2\1:
```

where `\1` holds the first word, `\2` - any number of spaces or tabs in between and `\3` - the second word. How to decide what number holds what pair of `\(\)` ? - count opening "`\(`" from the left.

Replacement Part of :substitute

Replacement part of the S&R has its own special characters which we are going to use to fix grammar:

#	Meaning	#	Meaning
&	the whole matched pattern	**\L**	the following characters are made lowercase
\0	the whole matched pattern	**\U**	the following characters are made uppercase
\1	the matched pattern in the first pair of \(\)	**\E**	end of \U and \L
\2	the matched pattern in the second pair of \(\)	**\e**	end of \U and \L
...	...	**\r**	split line in two at this point
\9	the matched pattern in the ninth pair of \(\)	**\l**	next character made lowercase
~	the previous substitute string	**\u**	next character made uppercase

Now the full S&R to correct non-capital words at the beginning of the sentences looks like

```
s:\([.!?]\)\s\+\([a-z]\):\1  \u\2:g
```

We have corrected our grammar and as an extra job we replaced variable number of spaces between punctuation and the first letter of the next sentence with exactly two spaces.

4.6 Alternations

Using "`\|`" you can combine several expressions into one which matches any of its components. The first one matched will be used.

```
\(Date:\|Subject:\|From:\)\(\s.*\)
```

will parse various mail headings and their contents into \1 and \2, respectively. The thing to remember about VIM alternation that it is not *greedy.* It won't search for the longest possible match, it will use the first that matched. That means that the order of the items in the alternation is important!

Tip 3: Quick mapping to put \(\) in your pattern string

```
cmap ;\ \(\)<Left><Left>
```

4.7 Regexp Operator Precedence

As in arithmetic expressions, regular expressions are executed in a certain order of precedence. Here the table of precedence, from highest to lowest:

Precedence	Regexp	Description
1	**\(\)**	grouping
2	**\=,\+,*,\{n} etc.**	quantifiers
3	**abc\t\.\w**	sequence of characters/ metacharacters, not containing quantifiers or grouping operators
4	**\\|**	alternation

V. Global Command

5.1 Global search and execution

I want to introduce another quite useful and powerful Vim command which we're going to use later

:range* g[lobal][!]/*pattern*/*cmd

Execute the Ex command ***cmd*** (default "**:p**") on the lines within [***range***] where ***pattern*** matches. If ***pattern*** is preceded with a **!** - only where match **does not** occur.

The global commands work by first scanning through the [*range*] of of the lines and marking each line where a match occurs. In a second scan the [*cmd*] is executed for each marked line with its line number prepended. If a line is changed or deleted its mark disappears. The default for the [*range*] is the whole file.

Note: Ex commands are all commands you are entering on the Vim command line like `:s[ubstitute], :co[py] , :d[elete], :w[rite]` etc. Non-Ex commands (normal mode commands) can be also executed via

`:norm[al]`*`non-ex command`*

mechanism.

5.2 Examples

Some examples of `:global` usage:

```
:g/^$/ d
```

- delete all empty lines in a file

```
:g/^$/,/./-j
```

- reduce multiple blank lines to a single blank

```
:10,20g/^/ mo 10
```

- reverse the order of the lines starting from the line 10 up to the line 20.

Here is a modified example from Walter Zintz vi tutorial:

```
:'a,'b g/^Error/ . w >> errors.txt
```

- in the text block marked by `'a` and `'b` find all the lines starting with **Error** and copy (append) them to "errors.txt" file. **Note:** . (current line address) in front of the `w` is very important, omitting it will cause `:write` to write the whole file to "errors.txt" for every **Error** line found.

You can give multiple commands after `:global` using "|" as a separator. If you want to use "|' in an argument, precede it with "\'. Another example from Zintz tutorial:

```
:g/^Error:/ copy $ | s /Error/copy of the error/
```

- will copy all **Error** line to the end of the file and then make a substitution in the copied line. Without giving the line address `:s` will operate on the current line, which is the newly copied line.

```
:g/^Error:/ s /Error/copy of the error/ | copy $
```

- here the order is reversed: first modify the string then copy to the end.

VI. Examples

6.1 Tips and Techniques

A collection of some useful S&R tips:

(1) sent by Antonio Colombo:

"a simple regexp I use quite often to clean up a text: it drops the blanks at the end of the line:"

```
s:\s*$::
```

or (to avoid acting on all lines):

```
s:\s\+$::
```

6.2 Creating outline

For this example you need to know a bit of HTML. We want to make a table of contents out of `h1` and `h2` headings, which I will call majors and minors. HTML heading `h1` is a text enclosed by `<h1>` tags as in `<h1>Heading</h1>`.

(1) First let's make named anchors in all headings, i.e. put `<h1><a name="anchor">Heading</a></h1>` around all headings. The `"anchor`" is a unique identifier of this particular place in HTML document. The following S&R does exactly this:

```
:s:\(<h[12]>\)\(.*\s\+\([-a-zA-Z]\+\)\)\s*\(</h[12]>\):\1<a name="\3">\2</a>\4:
```

Explanation: the first pair of `\(\)` saves the opening tag (`h1` or `h2`) to the `\1`, the second pair saves all heading text before the closing tag, the third pair saves the last word in the heading which we will later use for "anchor" and the last pair saves the closing tag. The replacement is quite obvious - we just reconstruct a new "named" heading using `\1-\4` and link tag `<a>.`

(2) Now let's copy all headings to one place:

```
:%g/<h[12]>/ t$
```

This command searches our file for the lines starting with `<h1>` or `<h2>` and copies them to the end of the file. Now we have a bunch of lines like:

```
<h1><a name="anchor1">Heading1</a></h1>
<h2><a name="anchor2">Heading2</a></h2>
<h2><a name="anchor3">Heading3</a></h2>
.........................
<h1><a name="anchorN">HeadingN</a></h1>
```

First, we want to convert all `name="` to `href="#` in order to link table entries to their respective places in the text:

```
s:name=":href="#:
```

Second, we want our `h1` entries look different from `h2`. Let's define CSS classes "majorhead" and "minorhead" and do the following:

```
g/<h1>/ s:<a:& class="majorhead":
g/<h2>/ s:<a:& class="minorhead":
```

Now our entries look like:

```
<h1><a class="majorhead" name="anchor1">Heading1</a></h1>
<h2><a class="minorhead" name="anchor2">Heading2</a></h2>
```

We no longer need `h1` and `h2` tags:

```
s:<h[21]>::
```

and replace closing tags with breaklines `<br>`

```
s:/h[21]:br:
```

```
<a class="majorhead" name="anchor1">Heading1</a><br>
<a class="minorhead" name="anchor2">Heading2</a><br>
```

6.3 Working with Tables

Quite often you have to work with a text organized in tables/columns. Consider, for example, the following text

Asia	America	Africa	Europe
Africa	Europe	Europe	Africa
Europe	Asia	Europe	Europe

Suppose we want to change all "Europe" cells in the third column to "Asia":

```
:%s:\(\(\w\+\s\+\)\{2}\)Europe:\1Asia:
```

Asia	America	Africa	Europe
Africa	Europe	**Asia**	Africa
Europe	Asia	**Asia**	Europe

To swap the first and the last columns:

```
:%s:\(\w\+\)\(.*\s\+\)\(\w\+\)$:\3\2\1:
```

Europe	America	Africa	**Asia**
Africa	Europe	Europe	**Africa**
Europe	Asia	Europe	**Europe**

To be continued...

VII. Other Regexp Flavors

Here I would like to compare Vim's regexp implementation with others, in particular, Perl's. You can't talk about regular expressions without mentioning Perl.

(with a help from Steve Kirkendall) The main differences between Perl and Vim are:

- Perl doesn't require backslashes before most of its operators. Personally, I think it makes regexps more readable - the less backlashes are there the better.
- Perl allows you to convert any quantifier into a non-greedy version by adding an extra ? after it. So *? is a non-greedy *.
- Perl supports a lots of weird options that can be appended to the regexp, or even embedded in it.
- You can also embed variable names in a Perl regular expression. Perl replaces the name with its value; this is called "variable interpolation".

VIII. Links

Read VIM documentation about pattern and searching. To get this type ":help pattern" in VIM normal mode.

There are currently two books on the market that deal with VIM regular expressions:

- **"Learning the vi Editor"** by Linda Lamb and Arnold Robbins.
- **"vi Improved - VIM"** by Steve Oualline

Definitive reference on regular expressions is Jeffrey Friedl's **"Mastering Regular Expressions"** published by O'Reilly & Associates, but it mostly deals with Perl regular expressions. O'Reilly has one of the book chapters available online.