Regular Expressions

Sanjiv K. Bhatia
Department of Mathematics & Computer Science
University of Missouri – St. Louis
St. Louis, MO 63121
email: sanjiv@cs.umsl.edu

Abstract

Regular expressions provide a powerful tool for textual search in computers. The language of regular expressions forms the basis for many applications and languages in computers, including vi, sed, awk, and perl. In this article, I'll present the use of regular expressions to perform search and replace operations, using vi, sed, and awk as the applications.

1 Introduction

Regular expressions provide one of the most powerful tools in computer science to perform search and replace operations in textual data. Their power comes from the efficiency and flexibility afforded by allowing for variable information in search patterns. They can be extremely simple as just a string composed of letters and numbers. On the other extreme, they can be extremely complex in the form of strings that are entirely composed of special symbols that may not be easily decipherable. However, they follow simple rules of grammar that are not hard to learn. It takes only a little practice to master the complexities inherent in the set of special symbols. Furthermore, the set of special symbols is fairly small and a person with limited experience can start to use this language quickly.

A regular expression is loosely defined as a string of letters, numbers, and special symbols to describe one or more search strings. The search string may contain fixed or variable information. For example, you may want to search for the string *gray* in a text but you may not be sure whether the author has spelled the string as *gray* or *grey*, with both the spellings treated as correct by the spell checker. A regular expression allows you to specify

the variable information in search strings, while limiting the scope of search. Thus, both *gray* and *grey* are valid for search but *griy* is not.

Any person, who has ever performed a search for a string in a document on the computer, or even over the internet, has already been exposed to at least the simplest form of regular expressions – the literal string being searched for. The above statement holds regardless of whether the platform for search is running Unix, Windows, or some other environment. In this article, I am going to limit myself to Unix/Linux environments, and present the utilities from these platforms to illustrate the use and power of regular expressions.

2 Character Set for Regular Expressions

A regular expression is composed of two types of characters – literals and metacharacters. Literals are characters that represent themselves. In that respect, the entire alphabet – upper case and lower case characters in English – are treated as literals. In addition, the numerals act as literals most of the time.

Metacharacters are special characters that do not represent themselves. They are a means to connect the literals and provide for variable information in the regular expressions. It has been suggested that literals form the words in the language of regular expressions while metacharacters provide the grammar.

An important metacharacter is a delimiter for the regular expression. Delimiters enclose a regular expression at the beginning and end, and are always picked from the special characters. Effectively, a delimiter should not be considered to be a part of the regular expression. In the utilities such as `vi` and `sed`, a delimiter can be any special character as long as we use the same character on both ends of the expression. More often than not, people have used the character `/` as the delimiter. The reason for this choice is because `/` is used for search in `vi`. Some utilities, such as the `grep` family, do not require the use of delimiters to enclose the regular expression.

The set of metacharacters used in a regular expressions is fairly small and is enumerated as follows (notice the use of quoted metacharacters using \ character):

```
^  $  .  *  [  ]  \{  \}  \  \(  \)
```

In addition, the following metacharacters have been added to the above set to support extended regular expressions, such as the ones used in `egrep`

```
+  ?  |  (  )
```

The character `-` is considered to be a metacharacter only within the square brackets to indicate a range; otherwise, it is treated as a literal. Even in this case, the dash cannot be the first character and must be enclosed between the beginning and the end of range characters.

With the knowledge of character set, we can now start to develop the rules to create regular expressions for search.

3 Basic Regular Expressions

A simple regular expression is one that is composed of just the literals. Recall that a literal is defined as a character that represents itself. Thus, the character *a* is expected to match itself in any text string but it may or may not match its upper case version *A*. Similarly, a number *2* matches itself in most of the circumstances.

In the rest of this article, I'll enclose the regular expressions in a pair of `/` characters, and underline the matching portions in the text.

A simple regular expression `/ring/` will match the following patterns:

```
The phone is ringing
Joe proposed to Mary with a ring.
```

Similarly, the regular expression `/day is/` matches

```
Today is a Tuesday.
John said that his day is made.
```

The expression `/April 26, 1995/` matches

```
John was born on April 26, 1995.
```

As seen from above, a simple regular expression contains literals that can be alphabets, numbers, or special characters. In the next section, we'll illustrate the use of metacharacters to add variable information into the regular expressions.

4 Metacharacters for Variable Information

Metacharacters are used to include the variable information in text strings being searched for. They are invariably selected from the set of special characters enumerated in Section 2. In this section, I'll present the use of each metacharacter in a separate subsection.

4.1 Metacharacter .

The metacharacter representing period (.) is used as a variable to match *a single occurrence of* any character. Thus, the regular expression `/.ing/` matches any character followed by the string `ing` as follows:

```
John likes singing in the club.
Someone please stop the running child.
```

A sequence of periods will use multiple occurrences of characters. The expression `/lo..ed/` matches

```
Be careful with the loaded gun.
John entered the room and looked around.
```

A number of people use different separator for month, day, and year in dates. Thus, the expression `/04.26.2005/` will match any of the following formats of date:

```
Today's date is 04/26/2005
Today's date is 04-26-2005
Today's date is 04.26.2005
```

4.2 Metacharacter pair []

The pair of square brackets enclose a set of characters such that any one of the specified characters is considered to be a match. Thus, a vowel can be matched by the expression `/[aeiou]/`. We can look for a word that contains two consecutive vowels by specifying two sets, one after the other, as `/[aeiou][aeiou]/` to match

```
It is a beautiful day.
Be careful with the loaded gun.
```

We can use the metacharacter `-` to specify a range of characters. For example, the expression `/[a-z]/` specify all the lower case letters of the English alphabet. Similarly, the expression `/[0-5]/` specifies all the digits in the set `{0, 1, 2, 3, 4, 5}`. This set is also useful when we try to combine a search where we are unsure of the first character being in upper case or lower case. For example, `/[sS]unday/` matches

```
Sunday is a holiday.
```

The pair of square brackets are also used to indicate the absence of a set of characters. This is achieved by starting the set with the metacharacter `^`. The pattern `/[qQ][^u]/` searches for all the occurrences of `q` or `Q` that are not followed by a `u`, as

```
Iraq does not export much oil these days.
```

Within a pair of square brackets, some metacharacters, such as \, $, ., and *, lose their special meaning. Also, ^ is special only if it is the first character after the left square bracket.

4.3 Metacharacters ^ and $

The metacharacters ^ and $ are used to *anchor* a regular expression to the beginning and end of a line. A regular expression starting with ^ matches the expression at the beginning of the line. For example, `/^Iraq/` will match in a line only if it occurs at the beginning of a line, such as

```
Iraq is in news these days.
```

but it will not match the word if it is in the middle of the line, such as

```
There was no front page news on Iraq today.
```

In a similar manner, the metacharacter $, if present at the end of a regular expression, matches the expression at the end of the line. The expression `/Iraq$/` matches

```
That man is from Iraq
```

Notice that the string being searched for is not followed by any character.

4.4 Metacharacter *

The metacharacter asterisk (*) can follow a regular expression that represents a single character or even an arbitrary regular expression to represent a pattern of characters. For now, let us concentrate on single character regular expressions. The asterisk represents zero or more occurrences of the *preceding* regular expression. This is a departure from the interpretation of * in Unix shell where it is used as a wildcard for any number of characters regardless filenames without regard to preceding character.

As a first example, consider the expression `/ab*c/`. This expression will match the following strings

```
I can hardly follow John's accent.
Barney knows his abc.
While typing abbbbbbc, the key on b got stuck.
```

The first example matches `ac` for *zero occurrence* of `b` (character preceding *). The regular expression preceding * can be a set of characters enclosed in a pair of square brackets. This will match zero or more occurrences of any character described in the set. For

example, a number of times, we get emails that have been forwarded from one person to another. These emails generally have a set of characters preceding each line described as `> >> > > >` depending on the mailer settings. Since these characters occur at the beginning of line, we can anchor our regular expression as `/^[> ]*/` and delete them in `vi` by the command

```
:%s/^[> ]*//
```

A general rule to remember for metacharacter `*` is that it attempts to find the longest possible match in a line. The expression `/(.*)/` is used to indicate a set of characters enclosed in parentheses. However, if there are two sets of parentheses in the line, it encloses both of them as

```
Let us look for (this) and (that).
```

If we want to match just the first set of characters in parentheses, we have to use an expression such as `/([^)]*)/`, which looks for the left parenthesis, then any set of characters other than the right parenthesis, and finally, the right parenthesis.

```
Let us look for (this) and (that).
```

4.5 Searching for the Occurrence of Metacharacters

The metacharacters can be searched by quoting, using the character `\` to precede the metacharacter. The `\` makes the metacharacter lose its special meaning and it is treated as a literal. As an example, the period in a sentence can be searched by preceding it with `\`, such as `/i\.e\./`, and it matches

```
Let us look for the period, i.e. .
```

4.6 Range Metacharacters

It is also possible to specify a predetermined number of characters by using the pair of quoted braces enclosing a range. For example, a set of five consecutive `a`s can be described by `/a\{5\}/`, a set contained at least 3 `a`s can be described by `/a\{3,\}/`, and a set containing between 3 and 7 `a`s is described by `/a\{3,7\}`. The number to specify the range must be positive and less than 256.

4.7 Bracketing Expressions

The regular expressions can be bracketed by using a pair of quoted parentheses, also known as tagged metacharacters. This allows the search for a repeating regular expression that is made up of more than one character. For example, the multiple consecutive occurrences of string `abc` can be searched by `/\(abc\)*/`. This will match a sequence such as

```
John speaks only abcabcabc and nothing more.
```

The range metacharacter can be used to search for a fixed number of repetitions such as `/\(abc\)\{2\}/`

```
John speaks only abcabcabc and nothing more.
```

A regular expression to validate money amounts can be written as `/\$[0-9]\{1,3\}\(,[0-9]\{3\}\)*\.[0-9]\{2\}/`. This expression will match

```
The CEO earns $11,540,000.00.
The minimum wage should be $8.45.
```

5 Using Regular Expressions in Editing

Regular expressions make it easy to perform editing in large documents, specially when we are dealing with variable information. We have already see the use of regular expressions to perform search in documents. In this section, we'll see the simple search and replace operation, and then, move on to manipulate the variable information to our convenience.

The simple search and replace command in `vi` or `sed` os given by `s`. In `sed`, the command is given as

`s/`*search_pattern*`/`*replacement_pattern*`/`

where *search_pattern* and *replacement_pattern* are appropriate strings. A simple example to change a name is given by

```
s/Clinton/Bush/
```

Another example to replace some variable information with fixed information will be

```
s/[tT]hen/now/
```

In case we want to add some information before or after a variable pattern, we can do that easily by using the symbol ampersand (&). For example, if we have a set of lines as follows

```
John Smith makes 8.50 an hour.
Mary Jo Ellen makes 9.25 an hour.
```

and we want to put the `$` sign in front of the numbers, we can achieve that by using

```
s/[0-9]\.[0-9][0-9]/\$&/
```

resulting in

```
John Smith makes $8.50 an hour.
Mary Jo Ellen makes $9.25 an hour.
```

Finally, we can use quoted parentheses to *remember* variable information in the search pattern and put that in replacement string using quoted digits. Within a regular expression, a quoted digit (`\n`) takes on the value of the string that the regular expression beginning with the nth `\(` matched. Let us assume a list of names in the format

```
last-name, first-name initial
```

We can change it to the format `first-name initial last-name` by using the following `sed` command

```
s/\([^,]*\), \(.*\)/\2 \1/
```

Notice how the first quoted left parenthesis matches the second part in the replacement string `\1`. It should also be noted that quoted parentheses can be nested into another pair. This does not cause any ambiguity in identification as they are identified by only the opening left parenthesis.

Regular expressions in Unix also include some shortcuts to operate on a set of characters. Let us say that you have a document typed in all capital letters. You are required to change it such that the first letter of each word is capitalized but all the other letters are in lowercase. This is easily achieved by

```
s/\<\([A-Z]\)\([A-Z]\{1,\}\)/\1\L\2/g
```

This will change the sentence

```
THE GAS PRICE IS EXTREMELEY HIGH.
```

to

```
The Gas Price Is Extremeley High.
```

6 The Final Word

In this article, I have attempted to describe the use of regular expressions for search and replacement in the context of some Unix applications. Regular expressions provide a powerful language but as with any language, you have to practice using the language to become fluent in it. Of course, once you know the language well, it becomes almost second nature and you start wondering how you lived without it for so long.

A note of caution. Regular expressions are interpreted by an underlying regular expression engine. And it appears that the engines do not always follow a standard. The issue gets very frustrating when you are trying to learn something and some of the things do not work as expected. A while back, I was on a Sun machine and tried to use the expression `cat\(cat\)*` to search for a number of consecutive `cat` patterns in a document. I was perplexed when the expression worked on Linux but not on Sun. And then, I discovered that it works if I use the utilities specified in `/usr/xpg4/bin` but not in `/usr/bin` which happened to be the ones set as my default.

It is my hope that this article will lead you to an exciting journey of regular expressions. Happy regex'ing!!!

References

[1] J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, Sebastopol, CA, 2002.