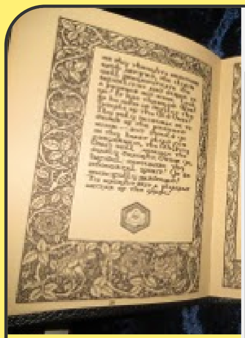

Regular Expressions

G+1 51

Table of Contents

Last updated - Mon Mar 7 20:22:51 EST 2011 - part of the Unix tutorials And then there's My blog

What is a Regular Expression?
The Structure of a Regular Expression
The Anchor Characters: ^ and $
Matching a character with a character set
Match any character with .
Specifying a Range of Characters with [...]
Exceptions in a character set
Repeating character sets with *
Matching a specific number of sets with \{ and \}
Matching words with \< and \>
Backreferences - Remembering patterns with \(, \) and \1
Potential Problems
Extended Regular Expressions
POSIX character sets
Perl Extensions
Thanks

Regular Expressions and Extended Pattern Matching

Bruce Barnett

Note that this was written in 1991, before Linux. In the 1980's, it was common to have different sets of regular expression features with different features. ed(1) was different from sed(1) which was different from vi(1), etc. Note that Sun went through every utility and forced each one to use one of two distinct regular expression libraries - regular or extended. I wrote this tutorial for Sun users, and some of the commands discussed are now obsolete. On Linux and other UNIX systems, you might find out that

awk
Reference
HTML
Magic
Search
About

some of these features are not implemented. Your mileage may vary.

Original version written in 1994 and published in the Sun Observer

What is a Regular Expression?

A regular expression is a set of characters that specify a pattern. The term "regular" has nothing to do with a high-fiber diet. It comes from a term used to describe grammars and formal languages.

Regular expressions are used when you want to search for specify lines of text containing a particular pattern. Most of the UNIX utilities operate on ASCII files a line at a time. Regular expressions search for patterns on a single line, and not for patterns that start on one line and end on another.

It is simple to search for a specific word or string of characters. Almost every editor on every computer system can do this. Regular expressions are more powerful and flexible. You can search for words of a certain size. You can search for a word with four or more vowels that end with an "s". Numbers, punctuation characters, you name it, a regular expression can find it. What happens once the program you are using find it is another matter. Some just search for the pattern. Others print out the line containing the pattern. Editors can replace the string with a new pattern. It all depends on the utility.

Regular expressions confuse people because they look a lot like the file matching patterns the shell uses. They even act the same way--almost. The square brackers are similar, and the asterisk acts similar to, but not identical to the asterisk in a regular expression. In particular, the Bourne shell, C shell, *find*, and *cpio* use file name matching patterns and not regular expressions.

Remember that shell meta-characters are expanded before the shell passes the arguments to the program. To prevent this expansion, the special characters in a regular expression must be quoted when passed as an option from the shell. You already know how to do this because I covered this topic in last month's tutorial.

The Structure of a Regular Expressio

There are three important parts to a regular expression. **Anchors** are used to specify the position of the pattern in relation to a line of text. **Character Sets** match one or more characters in a single position. **Modifiers** specify how many times the previous character set is repeated. A simple example that demonstrates all three parts is the regular expression "^#*". The up arrow is an anchor that indicates the beginning of the line. The character "#" is a simple character set that matches the single character "#". The asterisk is a modifier. In a regular expression it specifies that the previous character set can appear any number of times, including zero. This is a useless regular expression, as you will see shortly.

There are also two types of regular expressions: the "Basic" regular expression, and the "extended" regular expression. A few utilities like *awk* and *egrep* use the extended expression. Most use the "regular" regular expression. From now on, if I talk about a "regular expression," it describes a feature in both types.

Here is a table of the Solaris (around 1991) commands that allow you to specify regular expressions:

Utility	Regular Expression Type
vi	Basic
sed	Basic
grep	Basic
csplit	Basic
dbx	Basic
dbxtool	Basic
more	Basic
ed	Basic
expr	Basic
lex	Basic
pg	Basic
nl	Basic
rdist	Basic
awk	Extended
nawk	Extended
egrep	Extended
EMACS	EMACS Regular Expressions
PERL	PERL Regular Expressions

The Anchor Characters: ^ and $

Most UNIX text facilities are line oriented. Searching for patterns that span several lines is not easy to do. You see, the end of line character is not included in the block of text that is searched. It is a separator. Regular expressions examine the text between the separators. If you want to search for a pattern that is at one end or the other, you use *anchors*. The character "^" is the starting anchor, and the character "$" is the end anchor. The regular expression "^A" will match all lines that start with a capital A. The expression "A$" will match all lines that end with the capital A. If the anchor characters are not used at the proper end of the pattern, then they no longer act as anchors. That is, the "^" is only an anchor if it is the first character in a regular expression. The "$" is only an anchor if it is the last character. The expression "$1" does not have an anchor. Neither is "1^". If you need to match a "^" at the beginning of the line, or a "$" at the end of a line, you must *escape* the special characters with a backslash. Here is a summary:

Pattern	Matches
^A	"A" at the beginning of a line
A$	"A" at the end of a line
A^	"A^" anywhere on a line
$A	"$A" anywhere on a line
^^	"^" at the beginning of a line
$$	"$" at the end of a line

The use of "^" and "$" as indicators of the beginning or end of a line is a convention other utilities use. The *vi* editor uses these two characters as commands to go to the beginning or end of a line. The C shell uses "!^" to specify the first argument of the previous line, and "!$" is the last argument on the previous line.

It is one of those choices that other utilities go along with to maintain consistancy. For instance, "$" can refer to the last line of a file when using *ed* and *sed*. *Cat -e* marks end of lines with a "$". You might see it in other programs as well.

Matching a character with a characte set

The simplest character set is a character. The regular expression "the" contains three character sets: "t," "h" and "e". It will match any line with the string "the" inside it. This would also match the word "other". To prevent this, put spaces before and after the pattern: " the ". You can combine the string with an anchor. The pattern "^From: "

will match the lines of a mail message that identify the sender. Use this pattern with grep to print every address in your incoming mail box:

```
grep '^From: ' /usr/spool/mail/$USER
```

Some characters have a special meaning in regular expressions. If you want to search for such a character, escape it with a backslash.

Match any character with .

The character "." is one of those special meta-characters. By itself it will match any character, except the end-of-line character. The pattern that will match a line with a single characters is

```
^.$
```

Specifying a Range of Characters wit [...]

If you want to match specific characters, you can use the square brackets to identify the exact characters you are searching for. The pattern that will match any line of text that contains exactly one number is

```
^[0123456789]$
```

This is verbose. You can use the hyphen between two characters to specify a range:

```
^[0-9]$
```

You can intermix explicit characters with character ranges. This pattern will match a single character that is a letter, number, or underscore:

```
[A-Za-z0-9_]
```

Character sets can be combined by placing them next to each other. If you wanted to search for a word that

Started with a capital letter "T".
Was the first word on a line
The second letter was a lower case letter
Was exactly three letters long, and
The third letter was a vowel

the regular expression would be "^T[a-z][aeiou] ".

Exceptions in a character set

You can easily search for all characters except those in square brackets by putting a "^" as the first character after the "[". To match all characters except vowels use "[^aeiou]".

Like the anchors in places that can't be considered an anchor, the characters "]" and "-" do not have a special meaning if they directly follow "[". Here are some examples:

Regular Expression	Matches
[]	The characters "[]"
[0]	The character "0"
[0-9]	Any number
[^0-9]	Any character other than a number
[-0-9]	Any number or a "-"
[0-9-]	Any number or a "-"
[^-0-9]	Any character except a number or a "-"
[]0-9]	Any number or a "]"
[0-9]]	Any number followed by a "]"
[0-9-z]	Any number,
	or any character between "9" and "z".
[0-9\-a\]]	Any number, or
	a "-", a "a", or a "]"

Repeating character sets with *

The third part of a regular expression is the modifier. It is used to specify how may times you expect to see the previous character set. The special character "*" matches **zero or more** copies. That is, the regular expression "0*" matches **zero or more zeros**, while the expression "[0-9]*" matches zero or more numbers.

This explains why the pattern "^#*" is useless, as it matches any number of "#'s" at the beginning of the line, including **zero**. Therefore this will match every line, because every line starts with zero or more "#'s".

At first glance, it might seem that starting the count at zero is stupid. Not so. Looking for an unknown number of characters is very important. Suppose you wanted to look for a number at the beginning of a line, and there may or may not be spaces before the number. Just use "^ *" to match zero or more spaces at the beginning of the line. If you need to match one or more, just repeat the character set. That is, "[0-9]*" matches zero or more numbers, and "[0-9][0-9]*" matches one or more numbers.

Matching a specific number of sets with \{ and \}

You can continue the above technique if you want to specify a minimum number of character sets. You cannot specify a maximum number of sets with the "*" modifier. There is a special pattern you can use to specify the minimum and maximum number of repeats. This is done by putting those two numbers between "\{" and "\}". The backslashes deserve a special discussion. Normally a backslash **turns off** the special meaning for a character. A period is matched by a "\." and an asterisk is matched by a "*".

If a backslash is placed before a "<," ">," "{," "}," "(," ")," or before a digit, the backslash **turns on** a special meaning. This was done because these special functions were added late in the life of regular expressions. Changing the meaning of "{" would have broken old expressions. This is a horrible crime punishable by a year of hard labor writing COBOL programs. Instead, adding a backslash added functionality without breaking old programs. Rather than complain about the unsymmetry, view it as evolution.

Having convinced you that "\{" isn't a plot to confuse you, an example is in order. The regular expression to match 4, 5, 6, 7 or 8 lower case letters is

```
[a-z]\{4,8\}
```

Any numbers between 0 and 255 can be used. The second number may be omitted, which removes the upper limit. If the comma and the second number are omitted, the pattern must be duplicated the exact number of times specified by the first number.

You must remember that modifiers like "*" and "\{1,5\}" only act as modifiers if they follow a character set. If they were at the beginning of a pattern, they would not be a modifier. Here is a list of examples, and the exceptions:

Regular Expression	Matches
_	
*	Any line with an asterisk
*	Any line with an asterisk
\\	Any line with a backslash
^*	Any line starting with an asterisk
^A*	Any line
^A*	Any line starting with an "A*"

^AA*	Any line if it starts with one "A"
^AA*B	Any line with one or more "A"'s followed by a "B"
^A\{4,8\}B	Any line starting with 4, 5, 6, 7 or 8 "A"'s
	followed by a "B"
^A\{4,\}B	Any line starting with 4 or more "A"'s
	followed by a "B"
^A\{4\}B	Any line starting with "AAAAB"
\{4,8\}	Any line with "{4,8}"
A{4,8}	Any line with "A{4,8}"

Matching words with \< and \>

Searching for a word isn't quite as simple as it at first appears. The string "the" will match the word "other". You can put spaces before and after the letters and use this regular expression: " the ". However, this does not match words at the beginning or end of the line. And it does not match the case where there is a punctuation mark after the word.

There is an easy solution. The characters "\<" and "\>" are similar to the "^" and "$" anchors, as they don't occupy a position of a character. They do "anchor" the expression between to only match if it is on a word boundary. The pattern to search for the word "the" would be "\<[tT]he\>". The character before the "t" must be either a new line character, or anything except a letter, number, or underscore. The character after the "e" must also be a character other than a number, letter, or underscore or it could be the end of line character.

Backreferences - Remembering patterns with \(, \) and \1

Another pattern that requires a special mechanism is searching for repeated words. The expression "[a-z][a-z]" will match any two lower case letters. If you wanted to search for lines that had two adjoining identical letters, the above pattern wouldn't help. You need a way of remembering what you found, and seeing if the same pattern occurred again. You can mark part of a pattern using "\(" and "\)". You can recall the remembered pattern with "\" followed by a single digit. Therefore, to search for two identical letters, use "\([a-z]\)\1". You can have 9 different remembered patterns. Each occurrence of "\(" starts a new pattern. The regular expression that would match a 5 letter palindrome, (e.g. "radar"), would be

```
\([a-z]\)\([a-z]\)[a-z]\2\1
```

Potential Problems

That completes a discussion of the Basic regular expression. Before I discuss the extensions the extended expressions offer, I wanted to mention two potential problem areas.

The "\<" and "\>" characters were introduced in the *vi* editor. The other programs didn't have this ability at that time. Also the "\{*min*,*max*\}" modifier is new and earlier utilities didn't have this ability. This made it difficult for the novice user of regular expressions, because it seemed each utility has a different convention. Sun has retrofited the newest regular expression library to all of their programs, so they all have the same ability. If you try to use these newer features on other vendor's machines, you might find they don't work the same way.

The other potential point of confusion is the extent of the pattern matches. Regular expressions match the longest possible pattern. That is, the regular expression

```
A.*B
```

matches "AAB" as well as "AAAABBBBABCCCCBBBAAAB". This doesn't cause many problems using *grep*, because an oversight in a regular expression will just match more lines than desired. If you use *sed*, and your patterns get carried away, you may end up deleting more than you wanted too.

Extended Regular Expressions

Two programs use the extended regular expression: *egrep* and *awk*. With these extensions, those special characters preceded by a backslash no longer have the special meaning: "\{" , "\}", "\<", "\>", "\(", "\)" as well as the "*digit*". There is a very good reason for this, which I will delay explaining to build up suspense.

The character "?" matches 0 or 1 instances of the character set before, and the character "+" matches one or more copies of the character set. You can't use the \{ and \} in the extended regular expressions, but if you could, you might consider the "?" to be the same as "\{0,1\}" and the "+" to be the same as "\{1,\}".

By now, you are wondering why the extended regular expressions is even worth using. Except for two abbreviations, there are no advantages, and a lot of disadvantages. Therefore, examples would be useful.

The three important characters in the expanded regular expressions are "(", "|", and ")". Together, they let you match a **choice** of patterns. As an example, you can *egrep* to print all *From:* and *Subject:* lines from your incoming mail:

```
egrep '^(From|Subject): ' /usr/spool/mail/$USER
```

All lines starting with "From:" or "Subject:" will be printed. There is no easy way to do this with the Basic regular expressions. You could try "^[FS][ru][ob][mj]e*c*t*: " and hope you don't have any lines that start with "Sromeet:". Extended expressions don't have the "\<" and "\>" characters. You can compensate by using the alternation mechanism. Matching the word "the" in the beginning, middle, end of a sentence, or end of a line can be done with the extended regular expression:

```
(^| )the([^a-z]|$)
```

There are two choices before the word, a space or the beginining of a line. After the word, there must be something besides a lower case letter or else the end of the line. One extra bonus with extended regular expressions is the ability to use the "*," "+," and "?" modifiers after a "(...)" grouping. The following will match "a simple problem," "an easy problem," as well as "a problem".

```
egrep "a[n]? (simple|easy)? problem" data
```

I promised to explain why the backslash characters don't work in extended regular expressions. Well, perhaps the "\{...\}" and "\<...\>" could be added to the extended expressions. These are the newest addition to the regular expression family. They could be added, but this might confuse people if those characters are added and the "\(...\)" are not. And there is no way to add that functionality to the extended expressions without changing the current usage. Do you see why? It's quite simple. If "(" has a special meaning, then "\(" must be the ordinary character. This is the opposite of the Basic regular expressions, where "(" is ordinary, and "\(" is special. The usage of the parentheses is incompatable, and any change could break old programs.

If the extended expression used "(..|...)" as regular characters, and "\(...\|...\)" for specifying alternate patterns, then it is possible to have one set of regular expressions that has full functionality. This is exactly what GNU emacs does, by the way.

The rest of this is random notes.

Regular Expression	Class	Type	Meaning

_			
.	all	Character Set	A single character (except newline)
^	all	Anchor	Beginning of line
$	all	Anchor	End of line
[...]	all	Character Set	Range of characters
*	all	Modifier	zero or more duplicates
\<	Basic	Anchor	Beginning of word
\>	Basic	Anchor	End of word
\(..\)	Basic	Backreference	Remembers pattern
\1..\9	Basic	Reference	Recalls pattern
_+	Extended	Modifier	One or more duplicates
?	Extended	Modifier	Zero or one duplicate
\{M,N\}	Extended	Modifier	M to N Duplicates
(...|...)	Extended	Anchor	Shows alteration
_			
\(...\\|...\)	EMACS	Anchor	Shows alteration
\w	EMACS	Character set	Matches a letter in a word
\W	EMACS	Character set	Opposite of \w

POSIX character sets

POSIX added newer and more portable ways to search for character sets. Instead of using [a-zA-Z] you can replace 'a-zA-Z' with [:alpha:], or to be more complete. replace [a-zA-Z] with [[:alpha:]]. The advantage is that this will match international character sets. You can mix the old style and new POSIX styles, such as
grep '[1-9[:alpha:]]'
Here is the fill list

Character Group	Meaning
[:alnum:]	Alphanumeric
[:cntrl:]	Control Character
[:lower:]	Lower case character
[:space:]	Whitespace
[:alpha:]	Alphabetic
[:digit:]	Digit
[:print:]	Printable character
[:upper:]	Upper Case Character
[:blank:]	whitespace, tabs, etc.
[:graph:]	Printable and visible characters
[:punct:]	Punctuation
[:xdigit:]	Extended Digit

Note that some people use [[:alpha:]] as a notation, but the outer '[...]' specifies a character set.

Perl Extensions

Regular Expression		
Class	Type	Meaning
\t	Character Set	tab
\n	Character Set	newline
\r	Character Set	return
\f	Character Set	form
\a	Character Set	alarm
\e	Character Set	escape
\033	Character Set	octal
\x1B	Character Set	hex
\c[	Character Set	control
\l	Character Set	lowercase
\u	Character Set	uppercase
\L	Character Set	lowercase
\U	Character Set	uppercase
\E	Character Set	end
\Q	Character Set	quote
\w	Character Set	Match a "word" character
\W	Character Set	Match a non-word character
\s	Character Set	Match a whitespace character
\S	Character Set	Match a non-whitespace character
\d	Character Set	Match a digit character
\D	Character Set	Match a non-digit character
\b	Anchor	Match a word boundary
\B	Anchor	Match a non-(word boundary)
\A	Anchor	Match only at beginning of string
\Z	Anchor	Match only at EOS, or before newline
\z	Anchor	Match only at end of string
\G	Anchor	Match only where previous m//g left off

Example of PERL Extended, multi-line regular expression

```
    m{ \(
          (   # Start group
           [^()]+ # anything but '(' or ')'
          |   # or
            \( [^()]* \)
          )+  # end group
       \)
    }x
```

Thanks

Thanks to the following who spotted some errors

Charuhas Mehendale
Rounak Jain
Peter Renzland
Karl Eric Wenzel
Axel Schulze
Dennis Deters
Bryan Bergert

This document was translated by troff2html v0.21 on June 27, 2001.