

Know your regular expressions

Essential aids in building and testing regular expressions on UNIX systems

Michael Stutz

June 14, 2007

You can build and test regular expressions (regexps) on UNIX® systems in several ways. Discover the available tools and techniques that can help you learn how to construct regular expressions for various programs and languages.

The concept of *regular expressions* (regexps)—a notation for describing a pattern that matches a set of strings—is common across many programs and languages. These various regexp implementations differ to some degree in the finery of their details, but the principles for learning to build regexps are common for all.

This article describes some useful tools and techniques for learning to build and hone regexps across a range of UNIX® applications, including:

- [Highlighting matches](#)
- [Showing only the matches and not the lines](#)
- [Calling a wizard](#)
- [Studying docs](#)

Highlight matches in their context

When building a regexp, it helps to be able to see what strings the pattern matches in the context of a data set. Consider the four-line input text of [Listing 1](#) and the trivial regexp `t[a-z]` that matches a two-character pattern.

Listing 1. Four lines of sample text—and a regexp that matches them

```
$ cat midsummer
I know a bank where the wild thyme blows,
Where oxlips and the nodding violet grows,
Quite over-canopied with luscious woodbine,
With sweet musk-roses and with eglantine.
$ grep t[a-z] midsummer
I know a bank where the wild thyme blows,
Where oxlips and the nodding violet grows,
Quite over-canopied with luscious woodbine,
With sweet musk-roses and with eglantine.
$
```

Because it finds at least one match to its two-character pattern on every line, the `grep` command outputs every line in the input file. But which characters *exactly* on those lines of input did the regexp match?

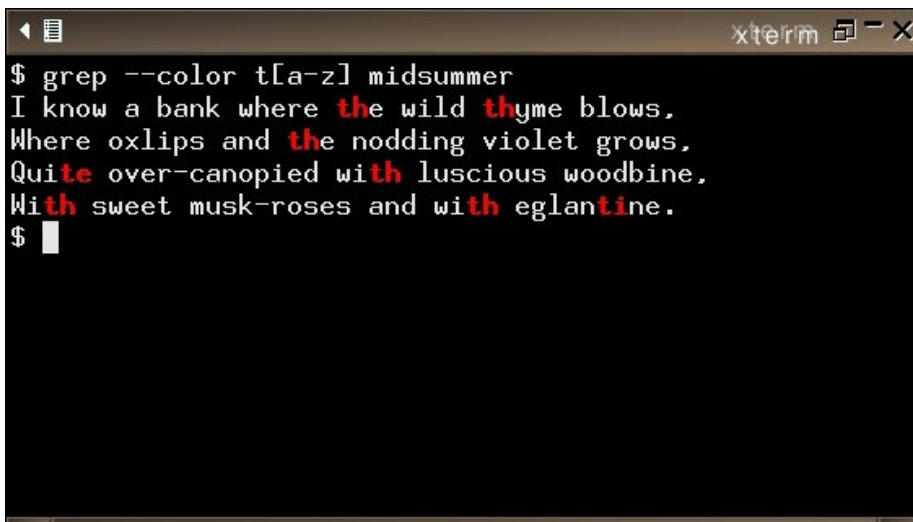
With a trivial regexp such as this one, it's easy to confidently eyeball it yourself. But as you build intricate regexps and have large datasets or input files, it can become considerably more difficult to know which string or strings a regexp might match. It's useful to be able to see exactly what text is being matched on each line. And one way to look at your regexps in context is to mark them in the output.

You can do that with several applications, including `grep`, `sed`, and Emacs.

Highlighting with `grep`

Some of the newer versions of `grep` (such as [GNU `grep`](#)) highlights the regexp in color when you use the `--color` option, as shown in [Figure 1](#).

Figure 1. Matched strings colorized in `grep`

A terminal window titled 'xterm' showing the command '\$ grep --color t[a-z] midsummer' and its output. The output consists of four lines of text from Shakespeare's 'A Midsummer Night's Dream'. The words 'the', 'thyme', 'the', 'Quite', 'With', and 'with' are highlighted in red. The prompt '\$' is shown at the beginning and end of the output.

```
$ grep --color t[a-z] midsummer
I know a bank where the wild thyme blows,
Where oxlips and the nodding violet grows,
Quite over-canopied with luscious woodbine,
With sweet musk-roses and with eglantine.
$
```

If your terminal supports color, this is a useful way to view exactly which strings your regexp is matching.

Highlighting with `sed`

You can also do regexp highlighting in `sed`, the stream editor. The `sed` command:

```
's/regexp/[&]/g'
```

outputs a copy of the input with all instances of *regexp* enclosed in brackets. [Listing 2](#) shows its output with the sample text.

Listing 2. Matched strings marked in sed

```
$ sed 's/t[a-z]/[&]/g' midsummer
I know a bank where [th]e wild [th]yme blows,
Where oxlips and [th]e nodding violet grows,
Qui[te] over-canopied wi[th] luscious woodbine,
Wi[th] sweet musk-roses and wi[th] eglan[ti]ne.
$
```

You can mark the regexps in other ways, too. If your input is a Groff document, you can add boldface to the regexp and send the document to `groff` for processing:

```
$ sed 's/t[a-z]/\\fB&\\fP/g' infile.roff | groff -
```

You can also write a short `sed` program to output matches in color. If your shell supports escape sequences, you can highlight all the regexps in the context of the file. Because escape sequences are cumbersome to type, you'll undoubtedly want to run it from a script, as shown in [Listing 3](#).

Listing 3. A sed script that highlights matched patterns in color

```
#!/bin/sh
# highlights regexp pattern in input file
# usage: hre regexp file
sed 's/'$1'/^[[34m&^[[37m/g' < $2
```

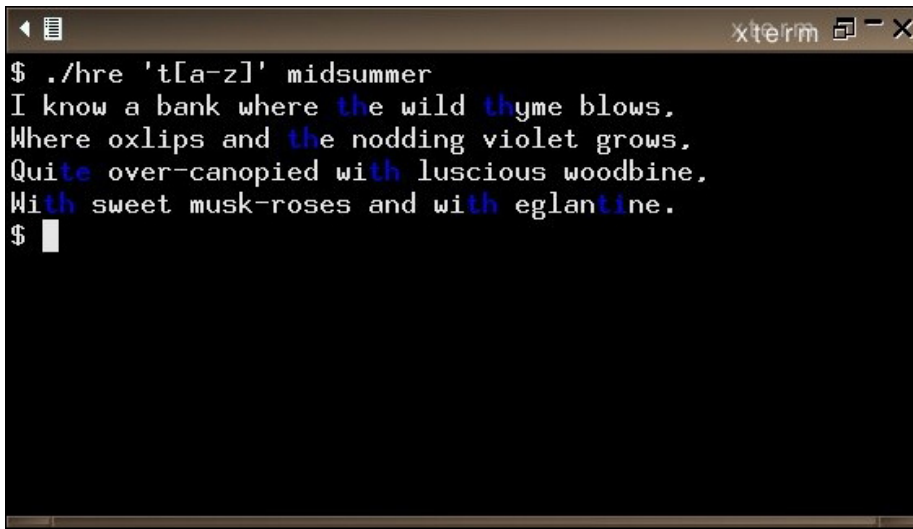
The `^[` that appears twice in the listing is a literal escape character, so you'll have to input this listing with an editor that supports entering literal characters, such as Emacs (where you'd type `c-q ESC` to enter it). The `34` and `37` are the Bash escape codes for specifying the colors blue and white, respectively.

To make the script executable, type:

```
$ chmod 744 hre
```

Then run it, as shown in [Figure 2](#).

Figure 2. Matched strings colored in sed

A terminal window titled 'xterm' showing the execution of a sed command. The command is `./hrc 't[a-z]' midsummer`. The output is a poem where words starting with a lowercase letter are highlighted in blue. The poem text is: `I know a bank where the wild thyme blows,
Where oxlips and the nodding violet grows,
Quite over-canopied with luscious woodbine,
With sweet musk-roses and with eglantine.` The terminal prompt `$` is visible at the beginning and end of the output.

```
$ ./hrc 't[a-z]' midsummer
I know a bank where the wild thyme blows,
Where oxlips and the nodding violet grows,
Quite over-canopied with luscious woodbine,
With sweet musk-roses and with eglantine.
$
```

While you can specify both the highlight and plain colors using this method, it has its caveats. The script shown in [Listing 3](#), for example, works only when the plain text of the terminal is white because it restores the text to that color. If your terminal uses a different color for plain text display, change the escape code in the script. (For example, `30` is black.)

Highlighting with Emacs

In new versions of the GNU Emacs editor, the `isearch-forward-regexp` and `isearch-backward-regexp` functions highlight all matches in the buffer. If you've installed a recent version of Emacs on your system, try it now:

1. Start Emacs by typing:

```
$ emacs midsummer
```

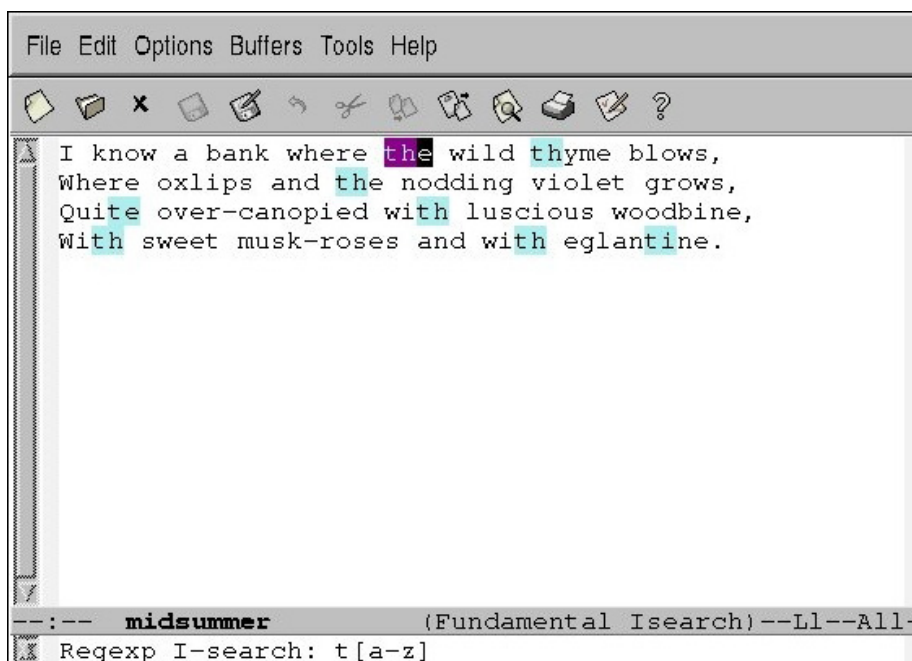
2. Type `M-x isearch-forward-regexp`.

The `M-x` is Emacs notation for the Meta-x combination, which you type on most systems either by pressing and holding the **Alt** key, pressing **X**, and then releasing both keys or by pressing the **Esc** key, releasing it, and then pressing the **X** key.

3. Type the regexp to search for: `t[a-z]`

Because the search is *incremental*, Emacs begins highlighting matches as you type a single character—in this case, when you press the **T** key, all the T characters in the buffer are highlighted. Notice that as soon as you begin to type the bracketed character list, the highlighting disappears, and Emacs reports in the minibuffer that it has insufficient input to show a match.

Your Emacs session should look like [Figure 3](#).

Figure 3. An Emacs buffer showing a regexp in the context

4. Type `c-x c-c` to exit Emacs.

You type this combination by pressing and holding the **Ctrl** key, pressing **X**, and then pressing and holding the **Ctrl** key and pressing **C**.

The `isearch-forward-regexp` and `isearch-backward-regexp` functions are typically bound to the `M-s-s` and `M-s-r` keystrokes. (To create them, press and hold the **Alt** key, the **Ctrl** key, and either the **S** or **R** key.)

Show only the matches, not the lines

There's another approach to the problem of pattern context, and that is to output only the matches themselves, not the entire lines in which they occur. There are ways to do this with `grep`, `sed`, and `perl`.

Show only the matches with `grep`

The `--only-matching` option (also `-o`) changes `grep`'s behavior so that it outputs not the entire lines containing a match to the regexp but *only those matches themselves*. As with the `--color` option described above, this feature appears in newer versions of some `grep` implementations, including GNU `grep`, which is open source and available for many operating systems.

This option is for collecting data that matches a regexp—it's great for harvesting IP addresses, URLs, names, e-mail addresses, words, and the like—but it's also a great way to learn regexps. For example, [Listing 4](#) shows how to use it to harvest all the words from the sample text of [Listing 1](#). It outputs each word, one to a line.

Listing 4. Harvesting all the words from the sample file

```
$ egrep -o '[A-Za-z]+' midsummer
```

```
I
know
a
bank
where
the
wild
thyme
blows
Where
oxlips
and
the
nodding
violet
grows
Quite
over
canopied
with
luscious
woodbine
With
sweet
musk
roses
and
with
eglantine
$
```

In fact, when you're constructing a particularly complicated regexp for a certain job, using this option is a simple way to test it to make sure that you've built it correctly. You can often see right away if your regexp needs fixing.

Say you want to output all the words in the test file containing the string `th`, and you've built the regexp shown in [Listing 5](#) to do that.

Listing 5. Outputting all words with "th," take one

```
$ egrep -o 'th[a-z]*' midsummer
the
thyme
the
th
th
th
th
$
```

Oh, that's not working. You can see right away that some of the matches in the output aren't words at all. Better try again: [Listing 6](#) takes into account any letters in the words that might come before the `th`.

Listing 6. Outputting all words with "th," take two

```
$ egrep -o '[a-z]*th[a-z]*' midsummer
the
thyme
the
with
ith
with
$
```

Much better, but still a little off. You see that one "ith" shows that the regexp didn't match uppercase letters. Rectify this by pulling out the `-i` option, as shown in [Listing 7](#).

Listing 7. Outputting all words with "th," take three

```
$ egrep -o -i '[a-z]*th[a-z]*' midsummer
the
thyme
the
with
With
with
```

That's it!

The use of `-o` and some test data are helpful in building regexps, because you might have assumed that the regexp worked as lines containing "th" were matched. But you didn't know that the expression was actually a little off.

Show only the matches with sed

You can do similar things in sed using the command:

```
s/.*\(\regexp\).*\/\1/p
```

to match a sed regexp. This command only outputs the matched patterns from the input, not the lines of the input that contain a match. However, it only outputs the *last* instance on a given line, as shown in [Listing 8](#).

Listing 8. Outputting only matched characters with sed

```
$ sed -n 's/.*\(\th[a-z]\).*\/\1/p' midsummer
thy
the
$ grep -o th[a-z] midsummer
the
thy
the
$
```

Show only the matches with Perl

Regexps are also popularly used in the Perl language, but Perl regexps are different from those you'd build using `grep`. The `ptest` tool lets you test Perl regexps. You can use this tool to familiarize yourself with the Perl-compatible regular expression (PCRE) library and to debug or test regexps that you build with it.

The regexp is enclosed in slash (/) characters as usual and can be followed with modifiers that alter the behavior of the search. Common regexp modifiers are provided in [Table 1](#).

Table 1. Common regexp modifiers for pcretest

Modifier	Description
<code>8</code>	This modifier supports Unicode (UTF-8) character sets.
<code>g</code>	This modifier searches for <i>global</i> matches (more than one on a line).
<code>i</code>	This modifier ignores differences in case.
<code>m</code>	This modifier searches over multiple lines.
<code>x</code>	This modifier uses extended Perl regexps.

Try running `pcretest` interactively, as shown in [Listing 9](#).

Listing 9. Testing a regexp with pcretest

```
$ pcretest
PCRE version 6.7 04-Jul-2006

re> /[a-z]*th[a-z]*/ig
data> With sweet musk-roses and with eglantine.
0: With
0: with
data>
$
```

You can also run `pcretest` with an input file. Input files contain a regexp to test on a single line followed by any number of lines of data to test. You can have multiple regexps and their respective data by separating them with an empty line; `pcretest` continues reading regexps and searching the following lines of data until it reaches end of file (EOF).

If you give the name of a second file, `pcretest` writes the output to that file. Otherwise, it writes to standard output, as shown in [Listing 10](#).

Listing 10. Running pcretest from an input file

```
$ cat midsummer.pre
/w[hi]|th/gi
I know a bank where the wild thyme blows,
Where oxlips and the nodding violet grows,
Quite over-canopied with luscious woodbine,
With sweet musk-roses and with eglantine.
$ pcretest midsummer.pre
PCRE version 6.7 04-Jul-2006

/w[hi]|th/gi
I know a bank where the wild thyme blows,
0: wh
0: th
0: wi
0: th
Where oxlips and the nodding violet grows,
0: wh
0: th
```



```
Quite over-canopied with luscious woodbine,  
0: wi  
0: th  
With sweet musk-roses and with eglantine.  
0: wi  
0: th  
0: wi  
0: th  
$
```

Call up a wizard

The `txt2regex` script is an interactive, cross-platform regexp "wizard" built for the Bash shell. When you run it, it asks you a series of questions about the pattern you want to match and then it builds valid regexps for any number of two dozen different applications:

- `awk`
- `ed`
- `egrep`
- `emacs`
- `expect`
- `find`
- `gawk`
- `grep`
- `javascript`
- `lex`
- `lisp`
- `mawk`
- `mysql`
- `ooo`
- `perl`
- `php`
- `postgres`
- `procmail`
- `python`
- `sed`
- `tcl`
- `vbscript`
- `vi`
- `vim`

Besides helping you interactively build regexps, `txt2regex` provides a concise summary of regexp syntax for various languages and applications, a list of "ready regexs" to match common patterns, and a handy chart of regexp metacharacters.

Build a regexp

To build a regexp for one or more of `txt2regex`'s [supported applications](#), give the names of those applications in a comma-delineated list as an argument to the `--prog` option.

Start by trying to build the trivial regexp given back in the [Highlight matches in their context](#) section, which matched the T character followed by a lowercase letter:

1. Start txt2regex and specify regexps for grep, sed, and Emacs:

```
$ txt2regex --prog grep,sed,emacs
```
2. You want to match the T character on any part of the line, not just at the beginning of the line, so type 2 to select "in any part of the line."
3. Type 2 again to select "a specific character" and then type `t` when asked which character to match.
 You now have to answer how many times you want to match it.
4. Type `1` to specify exactly once.
5. Match any lowercase letter by typing 6 to choose "a special combination" and then type `b` to match lowercase letters. Type `.` to exit the combination sub-menu.
6. Match the lowercase letter exactly once by typing `1`.

As you go through the procedure, txt2regex builds the regexp for each of the three chosen applications and displays them near the top of the screen. Now that you've selected exactly what you want, you can see the desired regexps for all three applications in [Figure 4](#).

Figure 4. Building a regexp with txt2regex

```
[.]quit [0]reset [*]color
[|]or [(]open group
Atxt2regex$
!! not supported
RegEx grep : t[a-z]
RegEx sed : t[a-z]
RegEx emacs: t[a-z]
.o0(22161) ( t :1)
[1-7]: 1
[1-9]:
followed by:
1) any character
2) a specific character
3) a literal string
4) an allowed characters list
5) a forbidden characters list
6) a special combination
7) a POSIX combination (locale aware)
8) a ready RegEX (not implemented)
9) anything
```

Type `..` to quit. The list of regexps will remain on your terminal.

Yes, all three of the regexps happen to be written as the identical `t[a-z]`, but that's only because this is a simple regexp and the three chosen applications have a similar regexp syntax. It won't always be the case that the regexps you build look the same for all chosen applications.

Say, for example, that you want to construct the two regexps you used in the [Show only the matches, not the lines](#) section. The first one was a single word of uppercase or lowercase letters:

1. Start txt2regex with no options:

```
$ txt2regex
```
2. Type 2 to match on any part of the line.
3. Type 6 to give a special combination and then type `a` and `b` to select all uppercase and lowercase letters.

4. Type `.` to return to the main menu and then type `4` to specify that it should be matched one or more times.

With no options, `txt2regex` defaults to build regexps for the `perl`, `php`, `postgres`, `python`, `sed`, and `vim` applications. When you run through the above, you'll find that the first four applications use the same regexp you used with `grep` in [Listing 4](#), but the regexps for `sed` and `vim` are just a little different. That's because these applications use a slightly different metacharacter notation, [as described below](#).

Again, type `..` to exit the program; the regexps for the various applications will remain listed on your terminal. You can use them as displayed or edit them to refine them further. For instance, what about matching words containing an apostrophe (') character? *don't*, *who're*, *e'er*, *owner's*, *'cause*, *Joe's*, and so on? The regexp you've just built won't match them properly, as you'll see by [showing only the matches](#) (see [Listing 11](#)).

Listing 11. Improperly matching words with apostrophes

```
$ echo "Don't miss a word, just 'cause it's wrong." | egrep [A-Za-z]+
Don
t
miss
a
word
just
cause
it
s
wrong
$
```

You'll want to add the hyphen character to that bracketed list and demonstrate it again, as shown in [Listing 12](#). Notice that you have to quote the regexp now.

Listing 12. Properly matching words with apostrophes

```
$ echo "Don't miss a word, just 'cause it's wrong." | egrep "[A-Za-z']+"
Don't
miss
a
word
just
'cause
it's
wrong
$
```

The next regexp you used in the [Show only the matches, not the lines](#) section was for a single word containing "th" anywhere in the word. You had used regexps for `egrep`, `sed`, and `perl`; now try building it for plain `grep`:

1. Start `txt2regex`:

```
$ txt2regex
```

2. Type `/` to select the programs and then type `hkopqstx`. so that only a regexp for `grep` will be built.

3. Type `26ab.3` to select zero or more uppercase or lowercase letters anywhere on the line.
4. Type `2t12h1` to follow that with the characters T and H, each occurring exactly once.
5. Type `6ab.3` to follow that with zero or more uppercase or lowercase letters.
6. Type `..` to exit the program.

You can test the regexp you've just built, as shown in [Listing 13](#).

Listing 13. Matching a "th"-containing word with grep

```
$ grep -o [A-Za-z]*th[A-Za-z]* midsummer
the
thyme
the
with
With
with
$
```

Get a summary of regexp options

The `--showinfo` option just outputs a brief summary of information about building regexps for a particular program or language. Included in the output is the name and version of the application, regexp metacharacters, default escape metacharacter, metacharacters that require escaping by default, whether you can use tab characters in bracketed lists, and whether it supports the Portable Operating System Interface (POSIX) bracket expressions.

If you're a developer who works on several applications, this is a good way to get a quick summary of the regexp rules for a particular application, as shown in [Listing 14](#).

Listing 14. Getting a summary of regexp rules with txt2regex

```
$ txt2regex --showinfo javascript

  program  javascript: netscape-4.77
  metas    . [] [^] * + ? {} | ()
  esc meta \
  need esc \.*[{}|+?^$
  \t in [] YES
  [:POSIX:] NO

$ txt2regex --showinfo php

  program  php: 4.0.6
  metas    . [] [^] * + ? {} | ()
  esc meta \
  need esc \.*[{}|+?^$
  \t in [] YES
  [:POSIX:] YES

$
```

Get a ready regex

The `--make` option is described by its author as "a remedy for headaches." It outputs a regexp for one of several common patterns that are given as arguments, as listed in [Table 2](#).

Table 2. List of ready regexps available in txt2regex

Argument	Description
date	This argument matches dates in mm/dd/yyyy format, from 00/00/0000 to 99/99/9999.
date2	This argument matches dates in mm/dd/yyyy format, from 00/00/1000 to 19/39/2999.
date3	This argument matches dates in mm/dd/yyyy format, from 00/00/1000 to 12/31/2999.
hour	This argument matches time in hh:mm format, from 00:00 to 99:99.
hour2	This argument matches time in hh:mm format, from 00:00 to 29:59.
hour3	This argument matches time in hh:mm format, from 00:00 to 23:59.
number	This argument matches any positive or negative integer.
number2	This argument matches any positive or negative integer with an optional floating-point value.
number3	This argument matches any positive or negative integer with optional commas and an optional floating-point value.

For example, you can use this to get a ready regexp for any valid hour in military time, as shown in [Listing 15](#).

Listing 15. Getting a date regexp with txt2regex

```
$ txt2regex --make hour3
RegEx perl      : ([01][0-9]|2[0123]):[012345][0-9]
RegEx php       : ([01][0-9]|2[0123]):[012345][0-9]
RegEx postgres: ([01][0-9]|2[0123]):[012345][0-9]
RegEx python    : ([01][0-9]|2[0123]):[012345][0-9]
RegEx sed       : \([01][0-9]\|2[0123]\):[012345][0-9]
RegEx vim       : \([01][0-9]\|2[0123]\):[012345][0-9]
$
```

Know your metacharacters

Another useful txt2regex option is `--showmeta`, which outputs a table consisting of all the metacharacters used in building regexps for the supported applications. This option is shown in [Listing 16](#).

Listing 16. Displaying all metacharacters with txt2regex

```
$ txt2regex --showmeta
awk      +      ?      |      (
ed       \+     \?     \{\}    \|     \(\)
egrep    +      ?      |      (
emacs    +      ?      |      \(\)
expect   +      ?      |      (
find     +      ?      |      \(\)
gawk     +      ?      |      (
grep     \+     \?     \{\}    \|     \(\)
javascript +     ?     { }     |     (
```

lex	+	?	{}		()
lisp	+	?		\\	\\(\\)
mawk	+	?			()
mysql	+	?	{}		()
ooo	+	?	{}		()
perl	+	?	{}		()
php	+	?	{}		()
postgres	+	?	{}		()
procmail	+	?			()
python	+	?	{}		()
sed	\\+	\\?	\\{\\}	\\	\\(\\)
tcl	+	?			()
vbscript	+	?	{}		()
vi	\\{1\\}	\\{01\\}	\\{\\}		\\(\\)
vim	\\+	\\=	\\{}	\\	\\(\\)

NOTE: . [] [^] and * are the same on all programs.

\$

Study the docs

It pays to read the manuals. Your system might have a lot more documentation, including man pages, on building and using regexps than you might realize.

For example, `grep`, `sed`, and other tools like them have man pages that describe their regexp syntax and give examples. If you have GNU versions installed on your system, they are also likely to have Info documentation that contains much more information than the usual man pages—whole user manuals are sometimes installed there. For example, if you have GNU `sed` installed and you have the `info` binary, you can read the manual:

```
$ info sed
```

The Perl documentation (usually packaged and distributed separately from the main Perl source or binary package) contains a comprehensive man page on Perl regexps:

```
$ man perlre
```

And there's even more to that subject. The `pcrepattern` man page (distributed with the `pcretest` application, [as described above](#)) also describes Perl regexps.

Finally, the `regex` man page, available on many UNIX systems, provides information on building POSIX regexps. The information on this man page is taken from Henry Spencer's `regex` library (see [Related topics](#)).

Summary

A lot of tools and methods are available on UNIX systems for regexp building. You've just learned a few of the best of them.

These tools give powerful ways to craft, test, and hone regexps. Using these tools and techniques on a UNIX system is probably the best way to learn to build complex regexps. And it's also fun!

Related topics

- ["Speaking UNIX, Part 9: Regular expressions"](#) (Martin Streicher, developerWorks, April 2007): This article is a short primer on the ABCs of regular expressions.
- Check out other articles and tutorials written by Michael Stutz:
 - [Across developerWorks and IBM](#)
- Search the AIX® and UNIX library by topic:
 - [System administration](#)
 - [Application development](#)
 - [Performance](#)
 - [Porting](#)
 - [Security](#)
 - [Tips](#)
 - [Tools and utilities](#)
 - [Java™ technology](#)
 - [Linux®](#)
 - [Open source](#)
- [AIX and UNIX](#): The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- [IBM trial software](#): Build your next development project with software for download directly from developerWorks.
- [developerWorks technical events and webcasts](#): Stay current with developerWorks technical events and webcasts.
- [Podcasts](#): Tune in and catch up with IBM technical experts.
- [GNU Project Web site](#): Download a free copy of GNU grep for your operating system.
- [PCRE](#): Download a free copy of PCRE.
- [txt2regex script](#): Download a free copy of txt2regex script.
- [regex](#): Download a free copy of Henry Spencer's regular expression libraries.

© Copyright IBM Corporation 2007

(www.ibm.com/legal/copytrade.shtml)

[Trademarks](#)

(www.ibm.com/developerworks/ibm/trademarks/)