UNIX tips and tricks for a new user, Part 3:
Introducing filters and regular expressions

Using grep, sed, and awk

Skill Level: Intermediate

Tim McIntire (tm@timmcintire.net)
Consultant
Freelance Writer

12 May 2006

Discover the power of UNIX® filters. In this tutorial, you'll learn about the `grep` family in depth, including the syntax of regular expressions in many UNIX utilities. You'll also find out more about the stream editor, `sed`, as well as examine the `awk` pattern scanning language through examples and explanations.

Section 1. Before you start

Learn what to expect from this tutorial and how to get the most out of it.

About this series

This four-part tutorial series covers UNIX® from the ground up. The initial tutorial was a good brush-up for users who've been away from UNIX-like operating systems for some time. It's also useful for brand-new UNIX users coming from a Windows® background, because it uses references and comparisons to Windows. The second tutorial focused on the `vi` text editor, one of the most powerful (and mysterious) UNIX utilities available. This tutorial teaches you about the UNIX command-line filters that use regular expressions, including `grep`, `sed`, and `awk`.

About this tutorial

Unlocking the power behind UNIX command-line filters, such as `grep`, `sed`, and

awk, requires a solid understanding of regular expressions. This tutorial teaches new users what each of these utilities is capable of and how to use regular expressions to manipulate text. You'll start by using simple, playful examples with grep and progress to real-world examples of sed and awk.

Objectives

The objective of this tutorial is to make UNIX and Linux® users comfortable with three powerful command-line tools that can be used to search for and alter data quickly and efficiently. The beginning of the tutorial explains regular expressions that are used in the basic framework of many UNIX utilities (and programming languages). The following sections give examples of using regular expressions with grep, sed, and awk.

Prerequisites

You need a basic understanding of the command line for this tutorial. For some parts of this tutorial, a working understanding of how input and output is handled in UNIX with stdin, stdout, and pipe is also helpful.

System requirements

Access to a user account on any computer running any UNIX-like operating system is all you need to complete this tutorial. UNIX-like operating systems include the IBM AIX® operating system, Linux, Berkeley Software Distribution (BSD), Mac OS® X (using Terminal to access the command line), and many others.

Section 2. Regular expressions

A regular expression is a string of characters designed to search for or replace another string of characters. At first glance, this appears to be a pretty basic function. Most users are familiar with simple search and replace functions in just about every graphical text editor or word processing application. If this basic search and replace functionality is compared to a calculator, regular expressions can be compared to a full-fledged computer. The power of using regular expressions for search criteria should not be underestimated.

Filters that use regular expressions

Regular expressions are used by some of the most powerful UNIX-based command-line tools, including grep, sed, and awk (and some programming

languages, including Perl). Learning how to use regular expressions is a required step in moving from a basic user of the UNIX command line to a true power user. There are a few different versions of regular expression syntax and multiple versions of `grep`, `sed`, and `awk`, so this tutorial focuses on the most common constructs that are fairly standard across each implementation. Don't forget to reference the `man` pages your system to get specifics on syntax and command-line options.

The basics

Before exploring UNIX applications that use regular expressions, it is important to learn the basics. In this section, simply read along. Later you'll try some examples in `grep`.

A basic search

A regular expression uses strings of normal characters combined with special characters that indicate the criteria for the search. In the most basic case, no special characters are used at all. For instance, if you simply want to use the term `golf` as your search criteria, you type the following:

```
golf
```

This is a regular expression! It searches for all instances of the word `golf`. Regular expressions are case sensitive, so this search finds all instances of **g**`olf`, but it will not find instances of **G**`olf`.

Using brackets

To search for both **g**`olf` and **G**`olf`, you can use brackets (which are special characters in regular expressions) and list a string of individual characters to search for. This is akin to a search within a search (which is the magic behind regular expressions).

```
[Gg]olf
```

The same concept works for any list of characters -- it's not just used for case sensitivity. For instance, you might want to search for `golf` and `gelf`, a new sport that you made up:

```
g[oe]lf
```

The period

Now imagine you have a third sport, `gilf`, that you also want to check for. One method, using what you have learned so far, is to use `o`, `e`, and `i` in your search criteria. But as your search grows, you might want to find everything that starts with `g` and ends with `lf` with one character in between. To do this, use another special character, a period (.).

```
g.lf
```

This finds all strings that begin with `g` and end with `lf` with one character in between. To expand your search to all strings that begin with `g` and end with `f` with two characters in between, you can use two periods:

```
g..f
```

Section 3. Searching files with grep

Now that you have a basic understanding of the concept behind regular expressions, it's time to start using real-world examples so that you can see them in action. The first command-line application you'll experiment with is `grep`. `grep` actually gets its name from a regular expression, `g/RE/p`. `grep` is used to search for instances of a particular string in one or more files. By default, `grep` outputs each line that your search string appears in (as opposed to the search string by itself). If you are searching in multiple files, `grep` also outputs the filename the line was found in.

Create a file called grep.txt with the following text:

```
I like golf.
Golf is played on grass.
I created gilf.
```

The basic syntax for `grep` is:

```
grep REGULAREXPRESSION FILENAME(S)
```

A basic search

Now, go back to the first example of a regular expression: the word `golf` by itself. To use this expression with `grep`, type:

```
grep golf grep.txt
```

This command searches the grep.txt file for all instances of the string `golf` and outputs the lines containing the string. Your output should be this:

```
I like golf.
```

Using brackets

Next, experiment with some of the special characters discussed above. You can use brackets (a bracket expression) to indicate you'd like to search for `golf` and `Golf`:

```
grep [gG]olf grep.txt
```

The output looks like this:

```
I like golf.
Golf is played on grass.
```

The period

To search for `golf` and `gilf`, you could use brackets again. Instead, try using a period to indicate you'd like to search for any character between `g` and `lf`:

```
$grep g.lf grep.txt
```

The output looks like this:

```
I like golf.
I created gilf.
```

Searching for golf, Golf, and gilf

You've now found ways to get each variation of `golf`, but none of your searches have returned all three instances: `golf`, `Golf`, and `gilf`. Take a moment to think about how you would search for all three. There are multiple ways to do this. Here are a couple examples:

```
grep ..lf grep.txt
grep [gG][oi]lf grep.txt
```

Both methods return all three lines:

```
I like golf.
Golf is played on grass.
I created gilf.
```

Dash

Can you think of any more ways to do this? So far, you've only learned two special characters to use in a regular expression. This is just the beginning! Some special characters are used inside of other special characters. For instance, when you enclose a set of characters in brackets, you can use a dash (-) to search for a range

of characters. Add the following line to your text file:

```
What is g2lf?
```

With what you've learned so far, you know this line would be included in your search result if you used a regular expression like `g.lf` or `g[oi2]lf`. Using a period returns results with any character in that position; using `[oi2]` returns results with just `o i` or `2` in that position. You can implement a third method that includes more than just a few characters, but not every character, by using a dash:

```
grep g[a-z]lf
```

This method produces the following output:

```
I like golf.
I created gilf.
```

As you can see from the output, this method searches for any character that falls between `a` and `z` (in alphabetic order). This excludes strings with numbers or symbols in between `g` and `lf` that are not really words and probably not part of your desired search criteria.

Dashes within brackets

You can also search for multiple character sequences by including additional sets within the brackets. For instance, to search for `a-z` and `A-Z`, use the following search:

```
grep g[a-zA-Z]lf
```

Caret

As your list of character sequences gets longer, you might find that it's easier to search by avoiding certain characters rather than specifying the characters you want to find. You can accomplish this by using the caret (^) before your search sequence (inside the brackets). That's quite a mouthful, but it should be easy to understand by following an example. Change your search to avoid numeric digits, but include all other characters by using this `grep` command:

```
grep g[^0-9]lf
```

This search is similar to the earlier search that found all alphabetic characters, but this one also returns characters, such as the number sign (`#`) and the dollar sign (`$`), that are not in the alphabet and are not in the number sequence you excluded.

Asterisk

The next special character to experiment with is the asterisk (*), which is one of several repetition operators. Most people are familiar with using the asterisk on the command line as a search criteria for filenames (a wildcard), but using an asterisk in a regular expression is quite different. An asterisk indicates the search item (previous character or bracket expression) can occur zero, once, or multiple times. To try this, add the following lines to the grep.txt file you have been working with:

```
This time the o is missing in glf.
Some people might say goolf.
But they would not say goilf.
```

The entire file should now look like this:

```
I like golf.
Golf is played on grass.
I created gilf.
What is g2lf?
This time the o is missing in glf.
Some people might say goolf.
But they would not say goilf.
```

Try using the asterisk after the `o` in `golf`:

```
grep go*lf grep.txt
```

Your search returns lines with the words `golf`, `glf`, and `goolf`:

```
I like golf.
This time the o is missing in glf.
Some people might say goolf.
```

Question mark

Another repetition operator is the question mark (?). The function of a question mark is similar to an asterisk, except the search item can occur zero or once. Multiple occurrences will not be matched. Try the search you just performed using a question mark instead of an asterisk:

```
grep go?lf grep.txt
```

As you can see, this time `golf` and `glf` are returned as matching results, but `goolf` is not because there are multiple instances of the search item `o` preceding the question mark:

```
I like golf.
This time the o is missing in glf.
```

Plus sign

The last of the general repetition operators is the plus sign (+). A plus sign will find when a search item occurs once or multiple times. Unlike the asterisk, at least one occurrence must be found to make a match. Try the following example:

```
grep go+lf grep.txt
```

This time, the search returns `golf` and `goolf`, but it does not return `glf` because no `o` is found:

```
I like golf.
Some people might say goolf.
```

Beginning-of-line and end-of-line anchors

The last special characters to learn about before moving on to `sed` are the beginning-of-line anchor, which is implemented with the caret, and the end-of-line anchor, which is implemented with the dollar sign. You might remember you used a caret earlier in the tutorial to negate a bracket expression. When a caret is used outside of brackets, it performs a completely different function. Putting a caret at the beginning of a regular expression tells the search to only operate at the beginning of a line. In other words, the first character in your regular expression (after the caret) must be a match for the first character on a new line for the search to match that line. Similarly, a dollar sign is put at the end of a regular expression to indicate you only want to return results that match at the end of a line. In other words, the last character in your regular expression (before the dollar sign) must be a match for the last character on a line for the search to match that line. To test this, add the following two lines to grep.txt:

```
golf has been a fine example
let's talk about something besides golf
```

Note that you shouldn't capitalize or punctuate `golf` for this test, because it will demonstrate a search for an identical word operating differently at the end of a line or the beginning of a line using anchors. To test the beginning-of-line anchor, type the following:

```
grep ^golf grep.txt
```

The output looks like this:

```
golf has been a fine example
```

To test the end-of-line anchor, use same search, but remove the caret and add a dollar sign after `golf`.

```
grep golf$ grep.txt
```

The output using the end-of-line anchor looks like this:

```
let's talk about something besides golf
```

Recap

Now you've learned the basics of regular expressions by using `grep` on the command line. Next, you'll learn to use `sed`, which not only searches for text but replaces it as well. First, here is a recap of what you've learned so far:

```
.   A period represents any single character
[]  Brackets enclose character sequences
-   A dash is used between characters to create a sequence (inside [])
^   A carrot is used to negate a sequence (inside [])
*   An asterisk searches for zero, one, or many instances of a search item
?   A question mark searches for zero or one instance of a search item
+   A plus sign searches for one or many instances of a search item
$   A dollar sign searches the end of a line
^   A carrot searches the beginning of a line
\   A backslash preceding a special character makes it a plain character (See the next
section.)
```

Section 4. Editing files with sed

`sed` is short for *stream editor*. The traditional, modern-day definition of a text editor is an interactive application that can be used to create and edit text files. `sed` is also a text editor, but it is a command-line utility instead of an interactive utility, which makes it an extremely powerful tool for batch editing. `sed` is commonly used in UNIX shell scripts to filter large sets of text files. In the first part of the tutorial, you used a small test file that discussed golf. To demonstrate the advanced capability of the `sed` editor, you'll use a small snippet of code that a developer might want to change in a batch process.

Copy and paste this text into a file named sed.txt:

```
system "echo 'project:$project' >> logfile";
system "echo 'version:$version' >> logfile";
system "echo 'optionalid:$optionalid' >> logfile";
system "echo 'nodes:$nodes' >> logfile";
system "echo 'threads:$threads' >> logfile";
```

Forward slash

All of the special characters explained earlier for use with `grep` also work in `sed`. To use `sed`, however, you must learn some additional syntax. A basic expression in `sed` is composed of four parts, separated by forward slashes (/). This is a common syntax for basic `sed` commands:

```
sed s/REGULAREXPRESSION/REPLACEMENTSTRING/flags INPUT_FILE
```

s -- Search and replace

The `s` indicates you want to execute a *search and replace*. The forward slashes are used to bind regular expressions in `sed`. For instance, if you simply want to replace the term `logfile` with `logfile.txt`, you would run the following command:

```
sed s/logfile/logfile.txt/ sed.txt
```

The output looks like this:

```
system "echo 'project:$project' >> logfile.txt";
system "echo 'version:$version' >> logfile.txt";
system "echo 'optionalid:$optionalid' >> logfile.txt";
system "echo 'nodes:$nodes' >> logfile.txt";
system "echo 'threads:$threads' >> logfile.txt";
```

An important point to note in this case is that `sed` will not actually change the contents of sed.txt. Instead, it sends the output to standard out. For these examples, you send the output to standard out so that you can immediately see the results of your actions.

For future reference, the output can be captured or sent into a new file. For example, to send the output to sed_new.txt, run this command:

```
sed s/logfile/logfile.txt/ sed.txt > sed_new.txt
```

Backslash

While you are learning about slashes, there is another very important special character to learn. The backslash (\) is called the escape character, because it *escapes* the next character from the regular expression interpretation. More simply, putting a backslash before a special character makes the character a plain item instead of a command item. This is important because many files, especially when you are writing code, make extensive use of the same characters that are used to execute a regular expression. In your sed.txt file, you'll notice that the dollar sign is used. If you want to replace `$project` but not `project`, you need to use the escape character in your search and replace:

```
sed s/\$project/\$project_name/ sed.txt
```

You can see in the output that `$project` was changed, but `project` was not.

```
system "echo 'project:$project_name' >> logfile";
system "echo 'version:$version' >> logfile";
system "echo 'optionalid:$optionalid' >> logfile";
system "echo 'nodes:$nodes' >> logfile";
system "echo 'threads:$threads' >> logfile";
```

Changing multiple instances of an item

This brings up another important feature in `sed`. What if you want to change both instances of `project`? From what you've learned so far, the logical answer would be to simply use `project` as your regular expression, but this isn't quite right. Go ahead and give it a try so that I can illustrate and explain the process:

```
sed s/project/project_name/ sed.txt
```

You can see in the output that the first instance of `project` was changed to `project_name`:

```
system "echo 'project_name:$project' >> logfile";
system "echo 'version:$version' >> logfile";
system "echo 'optionalid:$optionalid' >> logfile";
system "echo 'nodes:$nodes' >> logfile";
system "echo 'threads:$threads' >> logfile";
```

However, the second instance was not, even though it definitely matches your regular expression. You know from the first example that `sed` seems to change every matching string in its input, as opposed to just the first match, because it changed each instance of `logfile`.

The difference is that each instance of `logfile` was on a separate line, whereas there are two instances of `project` on the same line. Why does this make a difference? `sed` is implemented as a line editor. Each individual line is put into memory, one at a time, and operated on as a single unit. Keep this in mind when running `sed`, because all the command-line options are framed by this design philosophy (which allows most implementations of `sed` to have no file size limits relating to system memory). Each line is treated as a new execution of the `sed` command by default. Even though it didn't appear this way in the first example, each `sed` command replaces only the first instance of a matching string. You can, however, simply alter this with a `g` flag.

g flag

Execute the same `sed` command, but this time tack a `g` on the end:

```
sed s/project/project_name/g sed.txt
```

This time, both instances of `project` were changed to `project_name` on the first line:

```
system "echo 'project_name:$project_name' >> logfile";
system "echo 'version:$version' >> logfile";
system "echo 'optionalid:$optionalid' >> logfile";
system "echo 'nodes:$nodes' >> logfile";
system "echo 'threads:$threads' >> logfile";
```

You can remember `g` as being short for *global*.

Running a preliminary search

Another powerful feature of `sed` is running a preliminary search before the search and replace action to see if you are on a line where you want to execute your command. This is almost like doing a `grep` inside of a `sed`. In your example, you might want to change the log file for the `node` variable rather than grouping it with all of the other output. To do this, you want to change the string `logfile` to `logfile_nodes`, but only on the line that pertains to nodes. This command does just that:

```
sed  /nodes/s/logfile/logfile_nodes/ sed.txt
```

Here is the output:

```
system "echo 'project:$project' >> logfile";
system "echo 'version:$version' >> logfile";
system "echo 'optionalid:$optionalid' >> logfile";
system "echo 'nodes:$nodes' >> logfile_nodes";
system "echo 'threads:$threads' >> logfile";
```

Changing every string that ends with a colon

Now, try using some of what you learned about regular expressions while using `grep`, but in a `sed` command. You can change every string that ends with a colon by using the following regular expression inside of `sed`:

```
sed s/[a-z]*:/value:/g sed.txt
```

The output looks like this:

```
system "echo 'value:$project' >> logfile";
system "echo 'value:$version' >> logfile";
system "echo 'value:$optionalid' >> logfile";
system "echo 'value:$nodes' >> logfile";
system "echo 'value:$threads' >> logfile";
```

This is pretty cool, but it's not very logical. The reason it's not very logical is because you have the word `value` before all your variables with no way of knowing which variable is which. However, you can make this into a real-world example by using another feature of `sed`.

Ampersand

The ampersand (&) represents the string that was matched by your regular expression. In other words, if `[a-z]*:` turned out to be `project:` on a particular line, the ampersand would hold that value. This can be very useful. Take a look at this example:

```
sed s/[a-z]*:/new_\&/g sed.txt
```

This time, you altered each matching string, but you retained the identifier associated with each variable:

```
system "echo 'new_project:$project' >> logfile";
system "echo 'new_version:$version' >> logfile";
system "echo 'new_optionalid:$optionalid' >> logfile";
system "echo 'new_nodes:$nodes' >> logfile";
system "echo 'new_threads:$threads' >> logfile";
```

Executing multiple command sequences

You can also do multiple things at once with `sed`. To execute multiple command sequences at a time, you must use the `-e` flag before each expression. By default, `sed` interprets the first argument as an expression, but you need to be more explicit when running multiple commands, hence the `-e` flag. For example:

```
sed -e s/[a-z]*:/value:/g -e s/logfile/name/g sed.txt
```

You can see that in this case `sed` inserts `value:` and `name` in the appropriate places:

```
system "echo 'value:$project' >> name";
system "echo 'value:$version' >> name";
system "echo 'value:$optionalid' >> name";
system "echo 'value:$nodes' >> name";
system "echo 'value:$threads' >> name";
```

As you're starting to see, `sed` can be a very powerful tool for editing files in a large-scale batch process. In the previous example, you're operating on a single file, just like you did with `grep`. Don't forget that part of the power of these utilities is running them across multiple files, which you can do using wildcards or file lists in place of the single file you've been using in this tutorial.

Section 5. Using awk on the command line

This tutorial started with a basic explanation of regular expressions and subsequently introduced `grep` and `sed`. `grep` is a powerful search utility, while `sed` is an even more powerful search and replace utility. `awk` takes the next step, using regular expressions in a full-fledged command-line programming language. Just like `sed`, when `awk` is used on the command line, it takes line-based input. `awk` interprets one line of input at a time, but unlike `sed`, it processes each piece of input on the line as variables that can be used as input and output for inline code.

It should be noted that AWK (capitalized) is a full-fledged programming language that you can use to write scripts (as opposed to just being used on the command line), but this tutorial focuses on `awk`, which is the command-line utility that interprets AWK commands on the fly.

By the way, if anybody is reading this and trying to think of real-world uses for everything you have learned, I just used `grep` to search through some old code for good `awk` examples:

```
grep awk */*.pl
```

Most system administrators or programmers find uses for these tools on a daily basis. Here are a few lines of my output:

```
Edaemon/m_checkcurrentdisk.pl:$freespace = `awk '(NR==1) {print \$4 / 1024 / 1024}'
grep.tmp`;
Edaemon/m_getdatetime.pl:$month = `awk '(NR==1) {print \$2}' datetime.txt`;
Odaemon/odaemon.beowulf.dvd.pl:$filesize = `awk '(NR==1) {print \$1}' temp.txt`;
```

These are good examples because they show a very basic use of `awk`. For your first try, make it even simpler. For your tests with `awk`, create the following files in an empty directory (the content of each file is irrelevant, and they can be empty):

```
Screenshot_1.jpg
Screenshot_2.jpg
Screenshot_3.jpg
awk.txt
regular.txt
sed.txt
```

Using the output of ls as input into awk

By default, `awk` reads each line in an input file and separates the content into variables determined by blank spaces. In a very simple example, you could use the output of `ls` as input into `awk` and print the results. This example uses `ls` with the

pipe character (|) to send the output into `awk`:

```
ls | awk ' { print $1 } '
```

`awk` subsequently prints the first item on each line, which in this case is the only item on each line:

```
Screenshot_1.jpg
Screenshot_2.jpg
Screenshot_3.jpg
awk.txt
regular.txt
sed.txt
```

Using ls -l to generate multicolumn input for awk

That was really basic. For the next example, use `ls -l` to generate multicolumn input for `awk`:

```
ls -l
```

Implementations of `ls` vary a bit from system to system, but this is some example output:

```
total 432
-rw-rw-rw-   1 guest  guest  169074 Oct 15 14:51 Screenshot_1.jpg
-rw-rw-rw-   1 guest  guest   23956 Oct 15 20:56 Screenshot_2.jpg
-rw-rw-rw-   1 guest  guest   12066 Oct 15 20:57 Screenshot_3.jpg
-rw-r--r--   1 tuser  tuser     227 Oct 15 20:16 awk.txt
-rw-r--r--   1 tuser  tuser     233 Oct 15 19:35 regular.txt
-rw-r--r--   1 tuser  tuser     227 Oct 15 23:16 sed.txt
```

Note that the file owner is the third item on each line and the file name is the ninth item on each line (items are separated by spaces in `awk` by default). You can use `awk` to pull out the file owner and file name from this list by printing the third and ninth variable on each line. Here's how:

```
ls -l | awk ' { print $3 " " $9 } '
```

You'll notice the `print` command in `awk` has two quotes and an empty space in it. This is simply to print a space between the file owner and the file name in your output:

```
guest Screenshot_1.jpg
guest Screenshot_2.jpg
guest Screenshot_3.jpg
tuser awk.txt
tuser regular.txt
tuser sed.txt
```

You can put text in quotes between variables in an `awk` print statement.

Using regular expressions to specify lines

Now you've learned the basics of how to use `awk`, but isn't this tutorial about regular expressions? In `awk`, regular expressions are used heavily. The most common example is to precede an `awk` command with a regular expression that specifies the lines you want to operate on. As with `sed`, regular expressions in `awk` are encapsulated in forward slashes. For instance, if you only want to operate on files owned by `tuser`, you could use the following command:

```
ls -l | awk ' /tuser/ { print $3 " " $9 } '
```

The command produces this output:

```
tuser awk.txt
tuser regular.txt
tuser sed.txt
```

Changing file extensions

In another example, you might want to change the file extension of each of your text files without touching your image files. To do this, you'll want to separate your input variables with a period instead of a space, and then use a regular expression to indicate you only want to search for text files. To split variables based on a period, use the `-F` flag followed by the character you want to use in quotes. Try this example with the `awk` output piped to a shell (which will execute the `awk` generated commands):

```
s | awk -F"." ' /txt/ { print "mv " $1 "." $2 " " $1 ".doc" } ' | bash
```

A subsequent `ls -l` will show the new file names:

```
-rw-rw-rw-   1 guest  guest  169074 Oct 15 14:51 Screenshot_1.jpg
-rw-rw-rw-   1 guest  guest   23956 Oct 15 20:56 Screenshot_2.jpg
-rw-rw-rw-   1 guest  guest   12066 Oct 15 20:57 Screenshot_3.jpg
-rw-r--r--   1 tuser  tuser     227 Oct 15 20:16 awk.doc
-rw-r--r--   1 tuser  tuser     233 Oct 15 19:35 regular.doc
-rw-r--r--   1 tuser  tuser     227 Oct 15 23:16 sed.doc
```

Remember, these are the basics to get started with `awk`, but AWK is a full-fledged programming language capable of much more than the material presented in this tutorial. Take a look at the `awk man` page. If you want to learn even more, it would be wise to invest in a good book.

Section 6. Wrap-up

The examples in this tutorial should be enough to give you a basic understanding of UNIX filters using regular expressions and how they can be used on the command line. The three utilities used, `grep`, `sed`, and `awk`, have numerous built-in options and features that go well beyond the beginning lessons discussed in this tutorial. There are books wholly devoted to `sed` and `awk`. Search through the `man` page on `grep` to learn more about its power features.

If you feel like you've mastered the basics of regular expressions and want to take the next step, Perl is another great language that utilizes regular expressions to their full extent. Masters of Perl take joy in razor-thin, efficient lines of code that appear to be nonsensical strings of characters to unfamiliar users.

If you've followed each tutorial in this series, you've now learned basic file manipulation on the command line, how to use the `vi` text editor, and how to use command-line filters.

Keep an eye out for the next tutorial in this series, which will cover shell tricks and tips. In the meantime, learn all you can about regular expressions and the utilities in this tutorial. They allow you to turn long, complicated tasks into quick, elegant solutions you can be proud of!

Resources

Learn

- UNIX tips and tricks for a new user: Check out other parts in this series.
- *sed & awk* (O'Reilly, March 1997): This is a good resource for learning more about `sed` and `awk`.
- AWK: This Web site goes into more detail on AWK.
- "Common threads: Awk by example, Part 1" (developerWorks, December 2000): This article introduces `awk` and goes into more detail on `awk` as a programming language.
- AIX and UNIX articles: Check out other articles written by Tim McIntire.
- AIX and UNIX: Visit the developerWorks AIX and UNIX zone to expand your UNIX skills.
- New to AIX and UNIX: Visit the New to AIX and UNIX page to learn more about AIX and UNIX.
- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.
- AIX 5L Wiki: A collaborative environment for technical information related to AIX.
- Podcasts: Tune in and catch up with IBM technical experts.

Get products and technologies

- IBM trial software: Build your next development project with software for download directly from developerWorks.

Discuss

- Participate in the AIX and UNIX forums:
  - AIX 5L -- technical
  - AIX for Developers Forum
  - Cluster Systems Management
  - IBM Support Assistant
  - Performance Tools -- technical
  - Virtualization -- technical
  - More AIX and UNIX forums
- Participate in the developerWorks blogs and get involved in the developerWorks community.

About the author

Tim McIntire
Tim McIntire works as a consultant and co-founder of Cluster Corporation, a market leader in HPCC software, support, and consulting. He also contributes periodically to IBM developerWorks and Apple Developer Connection. Tim's research, conducted while leading the computer science effort at Scripps Institution of Oceanography's Digital Image Analysis Lab, has been published in a variety of journals, including *Concurrency and Computation* and *IEEE Transactions on Geoscience and Remote Sensing.* You can visit TimMcIntire.net to learn more.