1 Regular Expressions

As you spend more time in a UNIX environment, one concept which you will see again and again is regular expressions. A variety of UNIX tools and commands use regular expressions, unfortunately not always consistently. Nevertheless, regular expressions are a very powerful tool for searching and processing text, and even in their simplest form can make many seemingly complex tasks very easy. In this article, I'll define just what a regular expression is and show some simple ways to use them.

A regular expression is a way of describing a pattern of characters to a program so that it can either display or modify the occurrences of that pattern. The UNIX utility `grep` (which is an acronym for "global regular expression print") is the most basic UNIX tool which supports regular expressions; given a regular expression and one or more file names as arguments (in that order), `grep` will display to standard output each line of the file which contains the regular expression. (`grep` can also be used as a filter, by using it on the right hand side of a pipe and omitting the file name.) The simplest form of a regular expression is just a literal string; for example, if we wish to display each line of a file which contains the string `dog`, we can use the command "`grep dog filename`". A number of flags increase the utility of `grep`; they are displayed in the table below.

Table 1: Options for `grep`

Flag	Function	Flag	Function
`-i`	Ignore case	`-l`	List filenames only
`-n`	Include line numbers	`-v`	List non-matching lines
`-w`	Search for words		

Of course, it often becomes necessary to search for something more elusive than a literal string. To describe more general patterns, special characters are used. For example, in a regular expression, the period (`.`) represents exactly one occurrences of any character other than a newline. Thus, the regular expression `d.g` will be matched not only by `dog`, but by `prodigy` and `bandage` since each of them contain a `d` and a `g` separated by exactly one letter. The command "`grep d.g filename`" would display all those lines which contain such a pattern. To narrow the range of acceptable characters in a regular expression, a character class can be used. This is simply a list of one or more characters surrounded by square brackets [ ], and it is matched by only the characters within the brackets. If the first character in the bracket is a caret ^, then the character class is matched by anything **except** the characters in the brackets. Ranges of consecutive characters (such as a-z, A-Z or 0-9) can also be used in character classes, and have their obvious meanings. An asterisk (*) after a character or character class is matched by zero or more occurrences of that character class. To restrict matches based on the location of a pattern within a line, a caret (^) can be used to anchor a match to the beginning of a line, and a dollar sign ($) can be used to anchor a match to the end of a line.

In addition, escaped angle brackets (\< \>) can be used to specify word boundaries, as an alternative to the -w flag in grep. Thus the regular expression dog would be matched by dog, endogenous, bulldog, etc., while \<dog\> would be matched by only the word dog, and \<dog would be matched by dog and dogwood, but not bulldog.

It's very important to remember that many of the special characters which are used to form regular expressions also have special meaning to the shell. In fact, many of the features of regular expressions are used by the shell when it performs filename substitutions. Thus, when you submit a command to the shell which contains a regular expression, you should surround it in single quotes to protect the special characters from the shell. For example, the command grep \<dog\> filename will not do what you would expect; you need to quote the regular expression: grep '\<dog\>' filename. When using grep you should also remember that, if it fails to find a match for a regular expression it will not print any diagnostic error. So if you're using grep and see a message like "No match" or " ... not found", it usually means that the shell is trying to interpret the special characters, and that you need to enclose them in single quotes to proceed.

2 Regular Expressions with Editors

You can use regular expressions to find patterns in files from inside editors like vi and emacs. In vi, use the escape key to get into command mode, and enclose your regular expression in either slashes (/ /) for a forward search or backslashes (\ \) for a backward search. In emacs, the commands "search-forward-regexp" and "search-backward-regexp" provide similar functionality.

In addition, substitutions of text can also be performed based on the presence of regular expressions. For example, suppose we wish to remove leading zeroes from a file of numbers. A regular expression which will match a number with leading zeroes can be constructed as `\<00*`, that is a word boundary followed by one or more zeroes. Thus by substituting a null string for such a pattern, leading zeroes could be removed. This function is obtained using escape followed by an "s" command in vi, and with the command "replace-regexp" in `emacs`.

In the simple example above, we could perform the necessary change by focusing only on the zeroes in the number, which we wanted to eliminate anyway. A more challenging problem is one in which we need to describe a bit more of the surrounding territory of the pattern on which we want to focus. Suppose we have a file of containing both text and numbers in scientific notation (like `1.3e-5`), and we wish to change the small `e`'s of the scientific notation to capital `E`'s. It's not too hard to construct a regular expression which will find the numbers in question: `[.0-9]*e[-+0-9][0-9]*`[1]. However, in order to just change the `e` while preserving the rest of the number requires a new capability, namely tagging a sub-expression inside of a regular expression. In both `vi` and `emacs`, a tagged subexpression in a substitute command can be surrounded by escaped parentheses (`\( \)`) to tag it for future reference. Then, tagged patterns can be referred to as `\1`, `\2`, and so on, in the "to" portion of the substitute command. So, in `vi` for example, the necessary substitute command is `s/\([.0-9]*\)e\([-+0-9][0-9]*\)/\1E\2/g`. (The trailing `g` in the substitute command causes the substitution to take place for all occurrences within a line, not just the first.) A similar expression could be used with the "replace-regexp" command of emacs. Although expressions like this may appear complicated, they provide an efficient way of making complex contextual changes in text with a single command.

3 A Brief Warning

As I mentioned in the introduction to this article, regular expressions are not completely consistent from program to program. For example, the meaning

[1] Notice that, in the character class after the `e`, the dash (`-`) is the first character inside the brackets. This is necessary since it would otherwise be interpreted as part of a range of characters, like `a-z`.

of the asterisk (`*`) in the shell's filename expansion is different from that used by `grep` and other programs which support regular expressions. In addition, other versions of `grep` (like `fgrep` and `egrep`) support additional features beyond those described in this article. Programs like `perl` (which will be the topic of future articles in this series) also have many additional extensions to regular expressions beyond what I've described. The features described in this article should be supported by most programs which use regular expressions, but, as always, the online manual pages should be consulted to resolve any discrepancies.