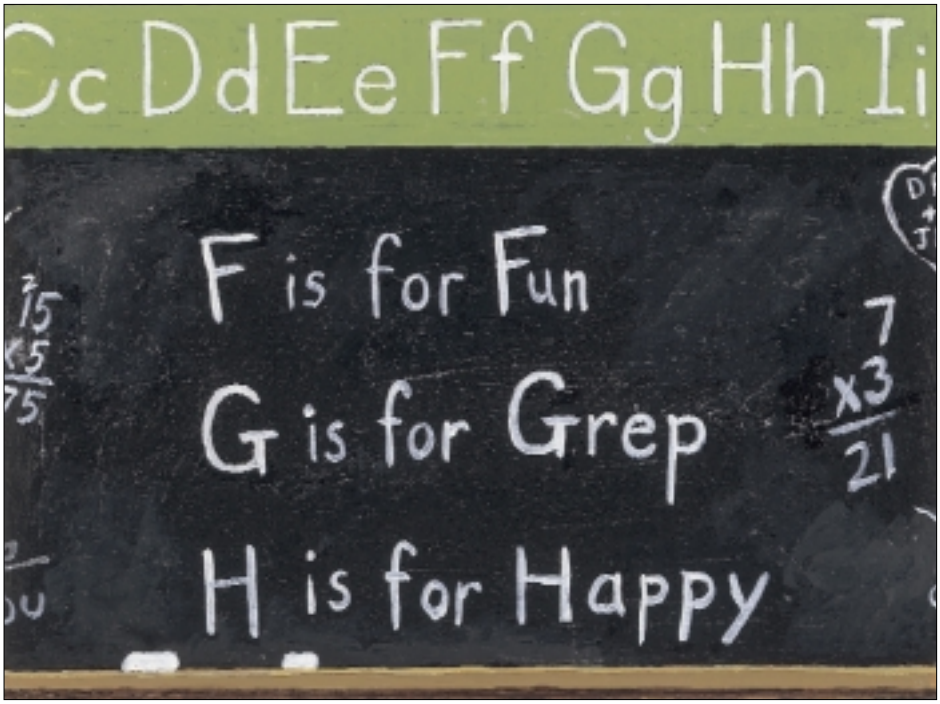
by Peter Collinson, **Hillside Systems**



DAVID FAUST

Grep is Fundamental

Several UNIX tools are so fundamental to my daily life that I almost take them for granted. The `grep` command is one of them. The name of the command is an acronym, Global Regular Expression Print, and it emerged from a common command sequence people typed regularly into the `ed` editor. To look for all the occurrences of a string in a file when editing it with `ed`, type

```
g/fred/p
```

which finds each line containing the string "fred" and displays it. The text between the slashes can be a *Regular Expression*, which is a way of expressing text patterns that are to be sought in the file. Of course, we usually type fixed strings because we are looking for something concrete.

Extracting this searching functionality into a command means you can search for the string in a named file or set of files and print any line that is found. Following on from my example,

```
$ grep fred *.ms
```

looks for `fred` in all the files in the current directory that end in `.ms`. In typical UNIX style, `grep` only outputs something when the line matches. If given more than one filename, it will print the contents of any matching line preceded by the name of the file where a match was found.

```
filesys.ms:ln: cannot create
fred:
grep.ms:string like fred.
korn.ms:% set fred=jim
```

Actually, there were 47 matching lines when I searched my article files, telling me that I am fond of using "fred" as an example string in my articles.

If you only search one file, then `grep` will drop the filename from its output. If you want to put it back, search the null device as well as the file using

```
grep fred onefile /dev/null
```

This little goody was one of the earliest pieces of feedback sent to me by a reader, and I've been using it ever since.

Using `grep` to search for strings answers the question "Can I find the text I am looking for in these files?" However, I probably use `grep` considerably more to restrict my output, reduce voluminous output to something that's readable and actually answers the question I am asking.

For example, let's say that I want to check that `sendmail` is running on the machine. I want to use the `ps` command to look at all the running processes and will type

```
$ ps -e
```

or perhaps

```
$ ps ax
```

depending on the flavor of `ps` I am using. On a typical machine, the command will print many lines of output

and, of course, I could look down the screen to find the `sendmail` process. It is, however, much quicker to use `grep` to select the output I want:

```
$ ps -e | grep sendmail
  354 ?        0:03 sendmail
```

The answer to my question is "Yes." The `sendmail` daemon is running and has process id 354. If `sendmail` is not running, then I will see no output.

I've used `grep` to reduce the amount of output, in this case to a single line, and this answers my question painlessly. Using `grep` gives me a result that is certain, so I don't have to look down several lines of output searching for something that may not be present. The idea that you should reduce the output of commands so you only obtain the answer to the question you are asking is applicable to many other situations with `grep` providing a "yes/no" filter for the output.

Options

The `grep` command has several useful options that modify its behavior. One that I use frequently is `-i` which forces a case independent search. This is particularly helpful when looking for matches in a set of documents where the word may be capitalized in the source. Another is `-w`, which looks for a "word." Again, this is more useful when searching a set of text documents and will pick complete words from the text. Your version of `grep` may not support this option.

I often use the `-l` (*ell* not *one*) option to `grep`. When searching a set of files, the `-l` option will restrict its output to the file names where a match is found. This is useful for providing editors with lists of files based on their content. For example,

```
$ vi `grep -l lookfor *.src`
```

runs `grep` looking for the string `lookfor` in the set of files ending with `.src` in the current directory. The list of found files is then supplied to `vi` via the *back-quote* operator where the output from command enclosed in the back quotes is used as the arguments to the `vi` command. The file handling commands in the editor can be used to step through the files making whatever changes are needed. If no files are found, then `vi` will be started with no arguments, and will open up an empty file. Obviously, you can use any editor in place of `vi`, as long as it deals sensibly with multiple file arguments. Some don't.

Another useful option is `-v` which reverses the sense of the command. Rather than printing the lines that match, it selects the lines that don't. I use this when refining a search, and use `grep` to find one set of lines and then pipe the result into `grep -v` to reduce the output.

If you've managed to get this far in this article, then I suspect that I've covered `grep` as it is used 99% of the time by most people. Few people will use even the simplest of the regular expression features that are supported by the command. Often it's not necessary to "get complicated" because life is too short. You can achieve the needed output by using several invocations of `grep` in a pipeline, each successively refining the output. However, getting the best from many UNIX tools demands a basic knowledge of regular expressions, and I don't apologize for returning to the topic again.

Regular Expressions

We've already seen that when you place a specific character sequence in a regular expression, then the search algorithm will look for that character sequence in the argument files. A match will be found when the complete string is found anywhere on any line in the source. Regular expressions provide a syntax for expressing complex matches, and we program the match using special characters in the regular expression called meta-characters.

Don't confuse the meta-characters used in `grep` and other programs that use regular expressions with the special characters employed by the shell to expand file names. Sometimes the same characters are used, but the meanings are different.

Also, when typing regular expressions into `grep` on the command line, we need to "get them past the shell." I have a habit of always wrapping single quotes around the regular expressions to ensure that they end up in the running `grep` command in an ungarbled form. Sometimes, in fact most times, quoting the program argument is not needed, but it's a good habit to get into.

The two simplest meta-characters to understand are `^` (caret) and `$` (dollar). The caret will match the start of a line in the source file, and the `$` will match the end of the line, so

```
$ grep '^fred' *
```

finds all the lines in files in the current directory that start with the word `fred`. The match is case sensitive.

```
$ grep '^fred$' *
```

finds all the lines that contain only the word `fred`.

A common idiom is `^$` - matching an empty line, so

```
$ grep -v '^$' file
```

will strip all the empty lines from the file and print the result to standard output.

If we want to look for the *characters* caret or dollar, then we must quote them using a backslash.

```
grep '^\^' file
```

will find all the lines in the file starting with a caret.

Variable Text

The real power of regular expressions comes from their ability to match variable or unknown text. If you are poking around in log files, you may want to match some known piece of text, followed by some text that's essentially unknown, followed again by some known text. Let's build this ability up slowly.

First, the period (.) in any character position in the regular expression will match any character that is found there. So

```
$ grep '^fre.$' /usr/dict/words
free
fret
```

looks for all the four-letter words in /usr/dict/words that start with fre followed by any character. I could have used the -w option instead of constraining the line to four characters with the caret and dollar syntax.

There's a tendency when you are looking for something, such as fred, to think of this as a single string that is to be matched. It's better to read and write the expression from left to right, thinking about what options are available to be matched in the particular character position. There will be some constant parts that are shown by characters and some variable parts described by meta-characters. So, the regular expression ^fre.$ is read as: the start of the line, followed by f, followed by r, followed by e, followed by any character, followed by the end of the line.

OK, so we can use the period to mean any character, but how do we express a variable character sequence? The asterisk (*) meta-character is used for this. When a * appears in the regular expression, the expression will match zero or more occurrences of the previous character in the regular expression. The "previous character" can be a meta-character or meta-character sequence. When it's a period, then the idiom .* will match sequences of any characters, so

```
$ grep '^ap.*tion$' /usr/dict/words
aphelion
apparition
apperception
application
apportion
apposition
apprehension
approbation
```

searches the dictionary for any word starting with ap and ending with tion. It's always important to recognize that the star operator will match zero occurences of the previous character and this feature can result in some confusion. It might seem reasonable to type

```
$ grep '^al*' /usr/dict/words
```

when looking for all the words starting with al and all in the dictionary, but if you try this, you'll find that it will match all the words starting with a. Reading the expression may help: it's the start of the line followed by a, followed by l repeated zero or more times. So we've allowed the match to have no l.

The regular expression really needed is

```
$ grep '^all*' /usr/dict/words
```

The egrep command allows for "extended regular expressions" and one of its extensions gets around this problem. The + (plus) character is used like the star, but matches *one* or more occurences of the previous character. However, egrep uses many more meta-characters and you may need to quote them if you want to match them in the source files, so beware when using egrep and expecting it to behave like a consistently extended grep.

The star meta-character always matches the longest string that it can find in the source resulting in what are apparently strange results from time to time. It's rarely a problem in grep where you are looking for some text, but can be a problem in editors (or the expr command) where there are mechanisms to pick out the matched text and then operate on it.

Alternations

It's also common to want to look for alternate characters that may appear in certain character positions. If we want to match Fred or fred in the source file, for example, we need to be able to match F or f in the first character position. We use square brackets to enclose a list of characters to be matched so

```
$ grep '^[Ff]red' files
```

matches any line starting with either fred or Fred. I could have used the -i flag to achieve this, but that would be less precise. It would match all possible case variations of fred.

It's common to want to put ranges and boring to have to express them as a list. We can express ranges by using a minus character.

It's common to want to put ranges and boring to have to express them as a list. We can express ranges by using a minus character. Using this, we can match integral numbers at the start of the line by typing

```
$ grep '^[0-9][0-9]*' files
```

Notice that I've repeated the range to avoid the "zero or more" problem with the star meta-character. Normal quoting rules don't apply inside alternations, so you cannot use backslash followed by a minus to include the minus in the list of characters to be matched. The trick is to add the minus character either immediately after the first square bracket or immediately before the terminating square bracket. The same rule applies when you want to include square brackets in the alternation list.

You sometimes want to match inverses of lists, and do this by adding a caret after the first square bracket. So

```
$ grep '^[^0-9A-Za-z]' files
```

finds all the lines in the file that don't start with an alphanumeric character.

I also use alternations when writing regular expressions that include spaces–mostly for readability–which also lets me specify white space as a tab character or a space. It's hard to tell whether the blank parts of files contain spaces or tabs, and when matching white space don't get caught out by the zero or more rule.

Grep Flavors

Your machine has several flavors of `grep`. The common extra two are `fgrep` and `egrep`. The "Fast Grep" or `fgrep` command is supposedly faster because it doesn't handle regular expressions but does a straight string match. The program may have been faster when the command was originally created, but the standard `fgrep` is actually slower than the `grep` command.

I've already mentioned `egrep`, a version of `grep` that supports "extended" regular expressions. Perhaps the most useful extension is the ability to provide whole word alternatives or more complete alternative regular expressions that match the data on the source lines.

Pulling multiple lines from a file is difficult in regular `grep`. You can end up writing crazy alternation lists. For example, if we are scanning your mailbox looking for "From:" lines and "Subject:" lines, then you would have to write something like

```
$ grep '^[FS][ru][ob][mj][:e]' /var/mail/$USER
```

If you look hard, you'll see that this will match the start of the line, followed by either `From:` or `Subje`. It will do the job, but isn't particularly precise and may print more than you want. Worse, it is easy to get it wrong when you are typing it. With `egrep`, we can write these alternates out clearly:

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

The round brackets are used to indicate that the regular expression contains sub-expressions, and the vertical bar means "*or.*"

If you are looking for several strings or regular expressions at once, and that list is long, then both `fgrep` and `egrep` support an option that allows the strings to be matched (for `fgrep`) or the alternate regular expressions (for `egrep`) to be taken from a file. You supply the `-f` option for both commands and follow it by the filename where the list is to be found. It's also possible to supply alternatives to either command using the `-e` option.

Finally

The `grep` command has a long history and has inspired research into the speed of its algorithms. System V UNIX inherited some slightly different options to its `grep` commands, and on Solaris a special version (`/usr/xpg4/grep`) can supply compliance with XPG4 standards. The `grep` family are useful as a basic tool, looking for text in files, or as filters reducing the amount of information displayed.

I haven't covered all the options for `grep` and regular expressions in this article. Specifically, I've omitted all the possible meta-character forms and have stuck to the most commonly used subset. I have included the common idioms that I hope you will soon be typing without thinking about it. I recommend that you look at the several manual pages when using regular expressions, and especially check the manual pages when you are getting apparently anomalous results. One of the problems with regular expressions is that they are much easier to write than read or explain–write once read never.

Further Reading

The best book I have on `grep` is *UNIX Power Tools* by Jerry Peek, Tim O'Reilly & Mike Loukides, published by O'Reilly and Associates; 2nd Edition, August 1997, ISBN 1-56592-260-3.

***Peter Collinson** runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever ... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.