

# Desperately Seeking Something or Conducting Searches With Logic and Regular Expressions.

N.J.Gunton 7/97

<b>Grouping</b>	Individual
<b>Prerequisites</b>	Access to a Unix System
<b>Courses</b>	All
<b>Requirements</b>	Command-line interface to Unix
<b>Summary</b>	Provides an introduction to the use of logic and regular expressions in searching for files or strings. An introduction to grep & find. Simple shell scripts.
<b>Objectives</b>	To understand how to use simple regular expressions and the use of search facilities.

## 1. Regular Expressions

A common activity in the world of computers is looking for things, whether it is a file somewhere in a filesystem, any file containing a particular word or words, or items of interest out on the internet. The more specifically you define your search pattern the more specific are your results† A regular expression is a way of describing such a pattern. It may be a simple sequence of characters such as 'myfile' or 'text' (known as a string) or it may be a more complex expression containing alternatives such as [Mm]yfile which will match either 'Myfile' or 'myfile'. They provide a way of restricting a search to a set of alternatives. Many Unix tools and utilities recognize regular expressions. The syntax is more or less consistent across these tools. These utilities include such stalwarts as **sed**, **grep**, **awk**, **lex** and to a lesser extent **ls**. Perl also uses this syntax for its regular expressions and pattern matching.

You may already be familiar with pattern matching through the use of the wildcard operators when listing files. These are the symbols \* and ? used in many operating systems to represent *zero or more characters* and *exactly one character* respectively. For example the unix command

```
ls /directory-path/m*.*?
```

will list all files in the given directory that start with m, followed by zero or more characters, a dot and a suffix of exactly one character. It would match any of the following.

```
m.c          myfile.c    my_very_big_project.p  moretext.t
monsters.b   m1.a       m2.a                  m3.again.a
```

but will ignore filenames such as

```
M.c          Myfile.     myfile.    myfile.text
monsters     Monsters.b  m3.again  more.text
```

### Exercise 1:

Experiment with the following operators and the `ls` command. Either find a directory that has a good range of files in, such as `/usr/bin/`, or use the `touch` command to create some files in your home directory. The operators to use are

---

† Although if you define your search too tightly you may miss the item you're looking for.

Simple regular expression operators	
Operator	Description
*	0 or more characters
?	1 character <sup>†</sup>
[aAbBcC]	any single character from the list given
[a-cA-C]	any single character from the range given

Study the following commands and make a note of what you think the result will be. Then execute the commands and compare the result with your notes.

```
ls /usr/bin/[b-g]*o*
ls /usr/bin/[b-g]*o*[r-t]
ls /usr/bin/[b-g]*o[r-t]
```

What regular expression would match all of the following?

```
/usr/bin/libconvert   /usr/bin/mdir         /usr/bin/newalias
/usr/bin/listalias    /usr/bin/munchlist   /usr/bin/newaliases
```

## 2. find [path...] [expression]

find is not an easy beast to master. The man pages run to nearly 600 lines. Fortunately we can do quite a lot with just a few options. For example we can use the command

```
find /usr/bin/ -type f -name [b-g]*o*[rst] -print
```

to get a result similar to that of exercise 1. This seems like a lot of work when compared with the ls command above<sup>††</sup>. However it gives us a great deal more flexibility. A major difference is that find will search any subdirectories below the directory given in the path. For example

```
find /usr/ -type f -name g?n*p -print 2> /dev/null
```

will search /usr/ as well as any subdirectories such as /usr/local/ or /usr/lib/.

So what do all the options we have used mean? Stepping through the parameter list we have

The path to be searched. This can be an explicit path as above, or it can be the current directory. The current directory is represented, as usual, by a dot (.) as in

```
find . -name my*.c
```

which will search the current directory and any subdirectories.

The flag -type f allows you to restrict the search for a specific type of file. In this case we are looking for ordinary files. Other common flags are

```
d   directory.
l   symbolic link.
b   block special files (devices).
c   character special files (devices).
```

The flag -name is followed by the string, or regular expression, to be used in the search. A regular expression may need to be quoted on some systems to prevent the shell from treating it literally as in 'g?n\*p'.

-print is usually a default, it ensures that the output goes to the screen.

---

<sup>†</sup> Many tools that use ? treat it as matching zero or one character, but not ls which treats it as matching exactly one character.

<sup>††</sup> Why does find give us a different list to that produced by ls? Hint: try using ls -l.

The expression `2> /dev/null` just redirects all error messages into a black hole. This is useful if you are piping the output from `find` into a file as you don't want your nice file list full of messages like `find: cannot read dir /tmp/lost+found: Permission denied`. Unfortunately this only works under the bash shell and other likeminded shells. The default shell at UWE is `tsh`. This makes redirection of the error stream a little more complicated. viz

```
( find /usr/ -type f -name 'g?n*p' -print > ~/foundlist ) >& /dev/null
```

It's not often worth going to these lengths if you are using `csh` or `tsh` but then it's nice to know that it can be done for those odd occasions when you need to create tidy files. It's also handy when you want to run a process in background and don't want the error stream to be written to the screen.

Other useful options that help limit the search with `find` allow you to specify the size of the file(s) to searched for, files older(newer) than a given date or the owner. Some of the more useful are listed below. See the man pages for a complete description.

Search modifiers with find	
Option	Description
<b>-amin</b> <i>n</i>	file was last accessed <i>n</i> minutes ago
<b>-atime</b> <i>n</i>	file was last accessed 24* <i>n</i> hours ago
<b>-empty</b> †	file is empty and is either a ordinary file or a directory
<b>-exec</b>	*** see discussion below ***
<b>-iname</b> †	like <code>-name</code> but is case insensitive
<b>-mmin</b> <i>n</i>	file was last modified <i>n</i> minutes ago
<b>-mtime</b> <i>n</i>	file was last modified <i>n</i> *24 hours ago
<b>-ok</b>	*** see discussion below ***
<b>-path</b> <i>pattern</i> †	*** see discussion below ***
<b>-prune</b>	*** see discussion below ***
<b>-size</b> <i>n</i> [ <b>ck</b> ]	size of file is <i>n</i> 512 byte blocks ('c' bytes, 'k' kilobytes)
<b>-user</b> <i>uname</i>	true if file belongs to <i>uname</i>

From this you can see that `find` is of great use for systems administration and general housekeeping. It should be noted that the argument '*n*' in the table can be '*-n*', '*n*' or '*+n*' for less than *n*, exactly *n* and greater than *n*.

### Exercise 2:

Make a note of your answers to each of the following questions.

- How would you find all the man 1 files that have not been accessed for the last month?
- How many files are there in `/tmp/` that have not been accessed for more than 3 months, and that are greater than 5 kilobytes in size?
- Every process on a unix system is treated as if it was a special type of directory. A process has a sub-directory in the directory `/proc/`. The names of these subdirectories are the process ids (pid). Use `find` to discover how many of these processes are owned by you.
- Describe, in words, what the effect of the following is. Is there anything in your home directory that is matched by this?

```
find ~/ -type f -size -3k -mtime -25 -name '*' -print 2> /dev/null
```

---

† not available in all flavours of unix

## 2.1 find & boolean operators

We have already been using the and operator as it is implicit when there is a list of expressions. With

```
find expr1 expr2
```

the search will only succeed if *expr1* and *expr2* are true. If *expr1* is false then *expr2* will not be evaluated. The and operator can be made explicit by using `-a` as in

```
find expr1 -a expr2.
```

Parentheses, `()`, can be used to force the precedence of operator<sup>††</sup>. The parentheses will probably need to be escaped, or the shell will get confused by them. The way to do this is with a backslash as in

```
find ~/ -size +5k -a \( -name '*.out' -o -name '*.o' \) -print
```

which will find all files in your home directory and subdirectories that are greater than 5 kilobytes in size and have either a suffix of `.out` (executable) or `.o` (object files). Without the parentheses the precedence rules would bind the and operator to the first name. The result would be to list all the files that are larger than 5k and end in `.out` or files of any size that end in `.o`.

Logical operators in order of decreasing precedence	
operator	description
<code>( <i>expr</i> )</code>	Force precedence
<code>! <i>expr</i></code>	Negation. True if <i>expr</i> is false
<code><i>expr1 expr2</i></code>	And (implied)
<code><i>expr1 -a expr2</i></code>	Same as <code><i>expr1 expr2</i></code>
<code><i>expr1 -o expr2</i></code>	Or. <i>expr2</i> is not evaluated if <i>expr1</i> is true

### Exercise 2.1 :

- As a trainee systems administrator you have been asked to obtain a list of all the students files that are less than a week old or have been modified in the last week. You are to exclude all object files, postscript files ( they end in `.ps` ) and `a.out` files. Write down the expression that you would use. Note that you can't actually try it out as you don't have permission to read other students directories. You can log the results of the search to a file by using the `-fprint† filename` option instead of the `-print` option.
- Create a few dummy `.o` and `.out` files in your home directory. Test out the script that you have written by modifying it to search your home directory only. Check the resulting file list to ensure you have excluded the appropriate files.

## 2.2 Search pruning

It is sometimes necessary to search a directory tree which contains subdirectories that you do not want to search. This is where pruning comes in. Unfortunately as not all versions of `find` support all the options, the use of `-prune` is a bit tortuous under Solaris. A simple example taken from the man page

```
find . -name bin -prune -o -print
```

will print the names of all files in and below the current directory except any in `bin` directories. Literally if the name matches `bin` then prune it otherwise print it out. This is all very well but if we want to search for specific names as well as pruning directories then we have to do something like

```
find /usr/local \( -name bin -prune \) -o -name 'g*n' -print
```

which will find all files beginning with `'g'` and ending with `'n'` except those in any `bin` directory. Take note of the various escapes needed to prevent the shell from interpreting the parentheses etc.

---

<sup>††</sup> A and ( B or C ) will ensure that the or operation is done before the and.

<sup>†</sup> not on all versions of `find`. Use redirection as an alternative

On systems whose `find` command supports the `-path` option you can use

```
find /usr/local -path /usr/local/bin -prune -o -name 'g*n' -print
```

but as the Solaris version of `find` doesn't support it, nothing more will be said.

### exercise 2.2

- How many people with accounts on the unix system have the string smith or Smith in their user id? Create a file of the results, which must exclude any members of staff. Make sure that you redirect errors to `/dev/null`. *see earlier discussion of redirection in the tsh shell*. This will take quite a while to run so consider running it in background and redirecting the output to a file.

### 2.3 Executing commands with find

`find` can execute commands on your behalf by means of the `-exec` command or the `-ok` command. The latter asks for confirmation before continuing. For example one could extend the earlier script that found all stale object files and `a.outs` so that it deleted them as it ran.

```
find $HOME \( -name '*.o' -o -name '*.out' \) -ok rm {} \;
```

Note the use of the environment variable `$HOME` and the escaped semicolon at the end.

- Create a directory in which to experiment with this as a mistake could cost you your file...(s) eg.

```
mkdir testbed
cd testbed
```

Use the `touch` command to create a mixture of files such as

```
touch this.o that.out the.other.o anything.some.suffix a.o b.o c.out
```

Now try out the `-ok` option and if you're happy with that then try out the `-exec` command. Use the `'.'` as the starting path to restrict it to your experimental test area. Once you are confident that it works as intended, you can use it to see if there are any such files lying around in your home directory.

- Suggest some other commands that it might be useful to execute in conjunction with `find`.

### 3. `grep` [-[AB] ] num ] [-[other options] ] [-e] pattern | [-f]file [files...] and complex regular expressions†

`grep` is a very useful search tool that will match patterns within a file or input stream. It can be used with a simple string or with a complex pattern. It uses a more extensive set of pattern matching characters than `find`. This includes matching a pattern only at the beginning or end of a line or matching only the beginning or end of a string. The biggest problem with `grep` is the variations between different versions. Every version of unix seems to come with a slightly different `grep`. One of the best is the GNU version which can often be found on most systems although it may be hidden away somewhere.

- Try using `find` but use parentheses and logical operators to exclude directories such as `/home` and `/sbin`. Run it in background saving the output to a file but exclude the error stream.

A simple use of `grep` is in finding specific processes out of the process list. Handy when you need to kill a process. Try

```
/usr/ucb/ps -aux | grep your user id
or
ps -fu your user id | grep x
```

The first will give you a long list of processes owned by you and the second will filter out all processes owned by you that have an 'x' in them. Another common and simple use of `grep` is checking if someone is logged on with

---

† Like all things unix, the available options vary according to flavour.

```
who | grep somename
```

Combining `grep` with `find` allows us to find a series of files and search them in turn with `grep`. Having to search through a whole stack of files for a specific keyword is a fairly common problem. The example below will search for ordinary files in the current directory and below. Each time it finds a file, it executes `grep` which then searches the file for the pattern that you have provided. If a match is found then, by default, `grep` will print the filename and the line containing the match to `stdout`. The reference to `/dev/null` is needed to force `grep` to print out the path and filename.

```
find . -type f -exec grep match /dev/null {} \;
```

- Use the above information to create your own version of the `apropos` command. Test it out using the man pages in `/usr/local/man/man1/`.
- How many files in your home directory contain the string 'float'?

You may have found that your version of `apropos` has one shortcoming in that it prints out every line in a given file that contains a match to *pattern*. This can be useful sometimes as it provides the context of the match, enabling you to decide if it is the file that you want. As an alternative you can provide the `-l` option to `grep` which will force it to print out the filename only. It does this as soon as it finds the first match in a file. This will execute much faster at the cost of less information. A fairly complete list of options is given below but check your local man pages for variations.

GREP general modifiers	
flag	action
<b>-num</b>	print matching lines with <i>num</i> leading & trailing lines
<b>-A num</b>	print <i>num</i> trailing context
<b>-B num</b>	print <i>num</i> leading context
<b>-c</b>	print a count of matching lines
<b>-e pattern</b>	use <i>pattern</i> as pattern, helps mask leading - in pattern
<b>-f file</b>	obtain the pattern from <i>file</i>
<b>-L</b>	print only the name of non-matching files
<b>-l</b>	print only the name of matching files
<b>-n</b>	prefix output with line number of match
<b>-s</b>	suppress error messages
<b>-v</b>	select only non-matching lines
<b>-w</b>	match whole words only
<b>-x</b>	exact match with whole line only

- Modify your version of `apropos` to match whole words only and print out both the number of matches within a file and the filename. What effect has this had on the output as compared with your original version?
- How many files are there in your home directory that are less than a week old and contain the whole word 'int'? Ensure that all matches are printed out with 2 lines of leading and trailing context.
- What is the disadvantage of using the `-c` option to `grep`? Is there any way to overcome this?

### 3.1 Complex pattern matching

Regular expressions can be considered as analogous to arithmetic expressions. Operators are used to combine smaller expressions into larger, more complex ones. The syntax falls into two categories, basic and extended. Some versions of `unix` provide two versions of `grep`, one for each type of regular expression, others provide options to control the type of expression.

This table of operators† can be used to build complex regular expressions. A fuller discussion can be found under the man entry for `regex`. Lexical analysers such as `lex` and `flex` and languages such as `PERL` use very similar pattern matching. Some characters have a special meaning to the shell and will need

protecting to prevent the shell from interpreting them. Typically these are ., ?, \*, [, and \. Others have a special meaning in regular expressions, such as ^ and \$. If you need to search for any of these characters then they must be preceded by a backslash \. The caret, ^, is only special in certain circumstances, as is the dollar sign, \$.

As an aside, some of the flags and options tend to get quite hard to remember for entering on the command line. One way round this is to turn them into shell scripts, which will then accept command line parameters such as the pattern to be matched. A brief introduction to shell scripts, along with exercises, is given later.

Pattern matching operators	
Operator	Description
<code>^<i>expr</i></code>	match at start of line only
<code><i>expr</i>\$</code>	match at end of line only
<code>.</code>	match any one character except newline
<code>[<i>aAbBcC</i>]</code>	any single character from the list given
<code>[^<i>aAbBcC</i>]</code>	any single character NOT in the list given
<code><i>A</i>?</code>	zero or one occurrence of <i>A</i>
<code><i>A</i>*</code>	zero or more occurrences of <i>A</i>
<code><i>A</i>+</code>	one or more occurrences of <i>A</i>
<code><i>A</i>{<i>m</i>}</code>	exactly <i>m</i> occurrences of <i>A</i>
<code><i>A</i>{<i>m</i>,}</code>	<i>m</i> or more occurrences of <i>A</i>
<code><i>A</i>{<i>m</i>,<i>n</i>}</code>	between <i>m</i> and <i>n</i> occurrences of <i>A</i>
<code><i>A</i>{<i>n</i>}</code>	between zero and <i>n</i> occurrences of <i>A</i>
<code>\&lt;<i>expr</i></code>	<i>expr</i> is matched at beginning of string only
<code>\&gt;<i>expr</i></code>	<i>expr</i> is matched at end of string only

For demonstration purposes, suppose that you need to find all the `public` and `private` declarations in your java source code files, but not the `protected` declarations. Having got this far through the worksheet you try the following.

```
( find $HOME -type f -name '*[Jj]ava*' -exec grep '[Pp][ru][^o]' \
    /dev/null {} \; ) >& /dev/null
```

You need to find all ordinary files with java somewhere in their names. The `[Jj]` allows for misspellings and looks impressive. You execute `grep` with a regular expression that matches any string that contain a 'P' or 'p' followed by either an 'r' or a 'u' and then followed by any character except an 'o'. This will match 'Private', 'private', 'Public' and 'public' but not 'protected' or 'Protected'. ( *The solitary \ hides the newline from the shell and allows you to spread the command over several lines. This is particularly useful in shell scripts as it adds legibility.* )

- Try out the above command to test how succesful this is in matching our requirements. How does this version improve the search results?

```
( find $HOME -type f -name '*[Jj]ava*' -exec grep '\<[Pp][ru][^o]' \
    /dev/null {} \; ) >& /dev/null
```

- What is the effect of replacing `*[Jj]ava*` with `*[Jj]ava'`?

Although this now gives us a reasonable result there are at least two things that could be improved. It would be nice if the full pathname and file name were printed only once for each file. The actual line number within the file would also be convenient. The current pattern also matches strings such as 'put', 'print', 'push' and many others.

The `-l` option to `grep` will print only the path and file names but then the actual line is no longer output. The `-h` will print the line but not the file name etc. The `-n` option will provide the line number. `-l` overrides the other two options and it would seem that the outcome wanted cannot be achieved. However `grep` can be called more than once as in

```
find $HOME -type f -name '*[Jj]ava' \  
-exec grep -l '\<[Pp][ru][^o]' /dev/null {} \; \  
-exec grep -hn '\<[Pp][ru][^o]' /dev/null {} \;
```

- Work out what this rather extreme pattern does and then write your own version that extracts all the 'public' variables in your java source code files. Can this script be simplified and still have the same results.

```
find $HOME -type f -name '*[Jj]ava' \  
-exec grep -l '\<[Pp][ru][^o][v1][a-zA-Z]*[ ][A-Z][a-zA-Z]*[(' /dev/null {} \; \  
-exec grep -hn '\<[Pp][ru][^o][v1][a-zA-Z]*[ ][A-Z][a-zA-Z]*[(' /dev/null {} \;
```

#### 4. Simple Shell Scripts

It is at this point that you might wonder whether it's worth the effort of developing command line scripts like the above and this is where shell scripts come into their own. Once they are written and tested they are like any other unix utility. They can take command line parameters etc.etc.† It is not the intention here to teach shell scripting but to give a taster of what can be done. The script opposite is designed to do a similar job to the exercises above and to demonstrate some of the constructs that are available for shell programming. The script takes a pattern and an optional path, then searches all ordinary files in the path for the pattern.

From the top, the line

```
#!/bin/sh
```

tells the system to use the `sh` shell. The first line of any shell script should dictate which shell to use. In practice shell scripts are always written in `sh`, as it works better than some others and because it is on all unix systems. Any other lines beginning with `#` are comment lines. The line

```
case $# in
```

is testing the number of parameters passed from the command line. `$#` holds the argument count, excluding the command name. `$1` `$2` etc. hold the first, second... parameters. The case statement works as expected, using the number of command line arguments to select which action to take. The asterisk denotes the default option. When passed a single argument, the script assumes it is a pattern to be matched and that you will search the current directory. It then lists every file in the directory

```
for fn in `ls`  
do
```

by calling `ls` and assigning each filename returned to the variable `fn`. Note the use of the grave accent ``` to quote the command. This causes the command to be executed and replaced by the output from the command. In this case `fn` will be replaced by the name of each file found in turn. The value of a shell variable is accessed by prefixing its name with `$` as in `$fn`.

The second option in the case statement is called when there are two arguments. It assumes that the first is the path to search and the second is the pattern to search for. The version shown saves the value of the current directory, tests to see that the directory exists and then changes to it before executing the script.

---

† In practice much of the work that was done with shell scripts is often now done with PERL but the principle is the same.

```
#!/bin/sh
#
# Grind will grind its way through all the files in a directory looking for
# files that contain a given regular expression. It will print the
# file name of any files containing the regular expression along with a
# count of the number of occurrences within a file.
#
#
#
#
#
case $# in
  1)
    for fn in `ls`
    do
      if [ -f $fn ] ; then
        grep -Gcl $1 $fn
      fi
    done ;;
  2)
    now=`pwd`;
    dir=${1}
    if [ -d $dir ] ; then
      echo "searching directory $dir"
    else
      echo "$dir does not exist or is not a directory";
      exit 0
    fi
    cd $dir;
    for fn in `ls`
    do
      if [ -f $fn ] ; then
        grep -Gcl $2 $fn
      fi
    done
    cd $now ;;
  *)
    echo "Usage : grind [expression] | [directory expression] " ;
    exit 0 ;;
esac
exit 0;
```

- A very simple shell script can be created as follows

```
who | grep $1
```

This command sequence will do it..

```
fred@spoon[1] cat > wg
> who | grep $1
> ^d
fred@spoon[2] chmod +x wg
fred@spoon[3]
```

Test this script and then try your hand at writing some other simple scripts.

- Copy the example script into a text editor and save it. Pay attention to the spacing around the [ and ] characters as it is important. change the mode of the file to make it executable and test it. For the adventurous...try modifying the script so that it uses `find` and takes three parameters, a pattern for `find`, a starting directory for `find` and a pattern for `grep`.