4.1 The `grep` family

We mentioned `grep` briefly in Chapter 1, and have used it in examples since then.

```
$ grep pattern filenames...
```

searches the named files or the standard input and prints each line that contains an instance of the *pattern*. `grep` is invaluable for finding occurrences of variables in programs or words in documents, or for selecting parts of the output of a program:

```
$ grep -n variable *.[ch]                Locate variable in C source
$ grep From $MAIL                        Print message headers in mailbox
$ grep From $MAIL | grep -v mary         Headers that didn't come from mary
$ grep -y mary $HOME/lib/phone-book      Find mary's phone number
$ who | grep mary                        See if mary is logged in
$ ls | grep -v temp                      Filenames that don't contain temp
```

The option `-n` prints line numbers, `-v` inverts the sense of the test, and `-y` makes lower case letters in the pattern match letters of either case in the file (upper case still matches only upper case).

In all the examples we've seen so far, `grep` has looked for ordinary strings of letters and numbers. But `grep` can actually search for much more complicated patterns: `grep` interprets expressions in a simple language for describing strings.

Technically, the patterns are a slightly restricted form of the string specifiers called *regular expressions*. `grep` interprets the same regular expressions as `ed`; in fact, `grep` was originally created (in an evening) by straightforward surgery on `ed`.

Regular expressions are specified by giving special meaning to certain characters, just like the `*`, etc., used by the shell. There are a few more metacharacters, and, regrettably, differences in meanings. Table 4.1 shows all the regular expression metacharacters, but we will review them briefly here.

The metacharacters `^` and `$` "anchor" the pattern to the beginning (`^`) or end (`$`) of the line. For example,

```
$ grep From $MAIL
```

locates lines containing `From` in your mailbox, but

```
$ grep '^From' $MAIL
```

prints lines that *begin* with `From`, which are more likely to be message header lines. Regular expression metacharacters overlap with shell metacharacters, so it's always a good idea to enclose `grep` patterns in single quotes.

`grep` supports *character classes* much like those in the shell, so `[a-z]` matches any lower case letter. But there are differences; if a `grep` character class begins with a circumflex `^`, the pattern matches any character *except* those in the class. Therefore, `[^0-9]` matches any non-digit. Also, in the shell a backslash protects `]` and `-` in a character class, but `grep` and `ed` require that these characters appear where their meaning is unambiguous. For example, `[][-]` (sic) matches either an opening or closing square bracket or a minus sign.

A period '`.`' is equivalent to the shell's `?`: it matches any character. (The period is probably the character with the most different meanings to different UNIX programs.) Here are a couple of examples:

```
$ ls -l | grep '^d'              List subdirectory names
$ ls -l | grep '^.......rw'      List files others can read and write
```

The '`^`' and seven periods match any seven characters at the beginning of the line, which when applied the output of `ls -l` means any permission string.

The *closure* operator `*` applies to the previous character or metacharacter (including a character class) in the expression, and collectively they match any number of successive matches of the character or metacharacter. For example, `x*` matches a sequence of `x`'s as long as possible, `[a-zA-Z]*` matches an alphabetic string, `.*` matches anything up to a newline, and `.*x` matches anything up to and including the *last* `x` on the line.

There are a couple of important things to note about closures. First, closure applies to only one character, so `xy*` matches an `x` followed by `y`'s, not a sequence like `xyxyxy`. Second, "any number" includes zero, so if you want at least one character to be matched, you must duplicate it. For example, to match a string of letters the correct expression is `[a-zA-Z][a-zA-Z]*` (a letter followed by zero or more letters). The shell's `*` filename matching character is similar to the regular expression `.*`.

No `grep` regular expression matches a newline; the expressions are applied to each line individually.

With regular expressions, `grep` is a simple programming language. For example, recall that the second field of the password file is the encrypted password. This command searches for users without passwords:

```
$ grep '^[^:]*::' /etc/passwd
```

The pattern is: beginning of line, any number of non-colons, double colon.

`grep` is actually the oldest of a family of programs, the other members of which are called `fgrep` and `egrep`. Their basic behavior is the same, but `fgrep` searches for many literal strings simultaneously, while `egrep` interprets true regular expressions — the same as `grep`, but with an "or" operator and parentheses to group expressions, explained below.

Both `fgrep` and `egrep` accept a `-f` option to specify a file from which to read the pattern. In the file, newlines separate patterns to be searched for in parallel. If there are words you habitually misspell, for example, you could check your documents for their occurrence by keeping them in a file, one per line, and using `fgrep`:

```
$ fgrep -f common-errors document
```

The regular expressions interpreted by egrep (also listed in Table 4.1) are the same as in grep, with a couple of additions. Parentheses can be used to group, so (xy)* matches any of the empty string, xy, xyxy, xyxyxy and so on. The vertical bar | is an "or" operator; today|tomorrow matches either today or tomorrow, as does to(day|morrow). Finally, there are two other closure operators in egrep, + and ?. The pattern x+ matches one or more x's, and x? matches zero or one x, but no more.

egrep is excellent at word games that involve searching the dictionary for words with special properties. Our dictionary is Webster's Second International, and is stored on-line as the list of words, one per line, without definitions. Your system may have /usr/dict/words, a smaller dictionary intended for checking spelling; look at it to check the format. Here's a pattern to find words that contain all five vowels in alphabetical order:

```
$ cat alphvowels
^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^aeiou]*$
$ egrep -f alphvowels /usr/dict/web2 | 3
abstemious          abstemiously        abstentious
acheilous           acheirous           acleistous
affectious          annelidous          arsenious
arterious           bacterious          caesious
facetious           facetiously         fracedinous
majestious
$
```

The pattern is not enclosed in quotes in the file alphvowels. When quotes are used to enclose egrep patterns, the shell protects the commands from interpretation but strips off the quotes; egrep never sees them. Since the file is not examined by the shell, however, quotes are *not* used around its contents. We could have used grep for this example, but because of the way egrep works, it is much faster when searching for patterns that include closures, especially when scanning large files.

As another example, to find all words of six or more letters that have the letters in alphabetical order:

```
$ cat monotonic
^a?b?c?d?e?f?g?h?i?j?k?l?m?n?o?p?q?r?s?t?u?v?w?x?y?z?$
$ egrep -f monotonic /usr/dict/web2 | grep '......' | 5
abdest     acknow     adipsy     agnosy     almost
befist     behint     beknow     bijoux     biopsy
chintz     dehors     dehort     deinos     dimpsy
egilops    ghosty
$
```

(Egilops is a disease that attacks wheat.) Notice the use of grep to filter the output of egrep.

Why are there three grep programs? fgrep interprets no metacharacters, but can look efficiently for thousands of words in parallel (once initialized, its running time is independent of the number of words), and thus is used primarily for tasks like bibliographic searches. The size of typical fgrep patterns is beyond the capacity of the algorithms used in grep and egrep. The distinction between grep and egrep is harder to justify. grep came much earlier, uses the regular expressions familiar from ed, and has tagged regular expressions and a wider set of options. egrep interprets more general expressions (except for tagging), and runs significantly faster (with speed independent of the pattern), but the standard version takes longer to start when the expression is complicated. A newer version exists that starts immediately, so egrep and grep could now be combined into a single pattern matching program.

Table 4.1: grep and egrep Regular Expressions (decreasing order of precedence)

c	any non-special character *c* matches itself
c	turn off any special meaning of character *c*
^	beginning of line
$	end of line
.	any single character
[...]	any one of characters in ...; ranges like a-z are legal
[^...]	any single character not in ...; ranges are legal
n	what the *n*'th \\(...\\) matched (grep only)
*r**	zero or more occurrences of *r*
r+	one or more occurrences of *r* (egrep only)
r?	zero or one occurrences of *r* (egrep only)
r1 r2	*r1* followed by *r2*
r1 | *r2*	*r1* or *r2* (egrep only)
\\(*r*\\)	tagged regular expression *r* (grep only); can be nested
(*r*)	regular expression *r* (egrep only); can be nested
	No regular expression matches a newline.

Exercise 4-1. Look up tagged regular expressions (\\(and \\)) in Appendix 1 or ed(1), and use grep to search for palindromes — words spelled the same backwards as forwards. Hint: write a different pattern for each length of word. □

Exercise 4-2. The structure of grep is to read a single line, check for a match, then loop. How would grep be affected if regular expressions could match newlines? □