Advanced Editing on UNIX

Brian W. Kernighan

(Updated for 4.3BSD by Mark Seiden)

ABSTRACT

This paper is meant to help secretaries, typists and programmers to make effective use of the UNIX† facilities for preparing and editing text. It provides explanations and examples of

- special characters, line addressing and global commands in the editor ed;
- commands for ''cut and paste'' operations on files and parts of files, including the mv, cp, cat and rm commands, and the r, w, m and t commands of the editor;
- editing scripts and editor-based programs like grep and sed.

Although the treatment is aimed at non-programmers, new users with any background should find helpful hints on how to get their jobs done more easily.

1. INTRODUCTION

Although UNIX provides remarkably effective tools for text editing, that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, people who are not computer specialists _ typists, secretaries, casual users _ often use the system less effectively than they might. (There is a good argument that new users would better use their time learning a display editor, like *vi*, or perhaps a version of *emacs*, like *jove*, rather than an editor as ignorant of display terminals as *ed.)*

This document is intended as a sequel to *A Tutorial Introduction to the UNIX Text Editor* [1], providing explanations and examples of how to edit using *ed* with less effort. (You should also be familiar with the material in *UNIX For Beginners* [2].) Further information on all commands discussed here can be found in section 1 of the *The UNIX User's Manual* [3].

Examples are based on observations of users and the difficulties they encounter. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying. There are also brief discussions of effective use of related tools, like those for file manipulation, and those based on ed, like grep and sed.

A word of caution. There is only one way to learn to use something, and that is to *use* it. Reading a description is no substitute for trying something. A paper like this one should give you ideas about what to try, but until you actually try something, you will not learn it.

† UNIX is a trademark of Bell Laboratories.

2. SPECIAL CHARACTERS

The editor ed is the primary interface to the system for many people, so it is worthwhile to know how to get the most out of ed for the least effort.

The next few sections will discuss shortcuts and labor-saving devices. Not all of these will be instantly useful to any one person, of course, but a few will be, and the others should give you ideas to store away for future use. And as always, until you try these things, they will remain theoretical knowledge, not something you have confidence in.

The List command 'l'

ed provides two commands for printing the contents of the lines you're editing. Most people are familiar with p, in combinations like 1,$p to print all the lines you're editing, or s/abc/def/p to change 'abc' to 'def' on the current line. Less familiar is the *list* command l (the letter '*l*'), which gives slightly more information than p. In particular, l makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, l will print each tab as $\underline{>}$ and each backspace as $\underline{<}$.† This

† These composite characters are created by overstriking a minus and a > or <, so they only appear as < or > on

makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The l command also 'folds' long lines for printing _ any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash , so you can tell it was folded. This is useful for printing long lines on short terminals.

Occasionally the l command will print in a line a string of numbers preceded by a backslash, such as 07 or 16. These combinations are used to make visible characters that normally don't print, like form feed or vertical tab or bell. Each such combination is a single character. When you see such characters, be wary _ they may have surprising meanings when printed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

The Substitute Command 's'

Most of the next few sections will be taken up with a discussion of the substitute command s. Since this is the command for changing the contents of individual lines, it probably has the most complexity of any ed command, and the most potential for effective use.

As the simplest place to begin, recall the meaning of a trailing g after a substitute command. With s/this/that/ and s/this/that/g the first one replaces the *first* 'this' on the line with 'that'. If there is more than one 'this' on the line, the second form with the trailing g changes *all* of them.

Either form of the s command can be followed by p or l to 'print' or 'list' (as described in the previous section) the contents of the line: s/this/that/p s/this/that/l s/this/that/gp s/this/that/gl are all legal, and mean slightly different things. Make sure you know what the differences are.

Of course, any s command can be preceded by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus 1,$s/mispell/misspell/ changes the *first* occurrence of 'mispell' to 'misspell' on every line of the file. But 1,$s/mispell/misspell/g changes *every* occurrence in every line (and this is more likely to be what you wanted in this particular case).

You should also notice that if you add a p or l to the end of any of these substitute commands, only the last line that got changed will be printed, not all the lines. We will talk later about how to print all the lines that were modified.

---

display terminals.

The Undo Command 'u'

Occasionally you will make a substitution in a line, only to realize too late that it was a ghastly mistake. The 'undo' command u lets you 'undo' the last substitution: the last line that was substituted can be restored to its previous state by typing the command u

The Metacharacter ''

As you have undoubtedly noticed when you use ed, certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. In the next several sections, we will talk about these special characters, which are often called 'metacharacters'.

The first one is the period ''. On the left side of a substitute command, or in a search with '/.../', '' stands for *any* single character. Thus the search /xy/ finds any line where 'x' and 'y' occur separated by a single character, as in x+y x−y xy xy and so on. (We will use to stand for a space whenever we need to make it visible.)

Since '' matches a single character, that gives you a way to deal with funny characters printed by l. Suppose you have a line that, when printed with the l command, appears as
.... th07is .... and you want to get rid of the 07 (which represents the bell character, by the way).

The most obvious solution is to try s/07// but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '' comes in handy. Since '07' really represents a single character, if we say s/this/this/ the job is done. The '' matches the mysterious character between the 'h' and the 'i', *whatever it is.*

Bear in mind that since '' matches any single character, the command s//,/ converts the first character on a line into a ',', which very often is not what you intended.

As is true of many characters in ed, the '' has several meanings, depending on its context. This line shows all three: s/// The first '' is a line number, the number of the line we are editing, which is called 'line dot'. (We will discuss line dot more in Section 3.) The second '' is a metacharacter that matches any single character on that line. The third '' is the only one that really is an honest literal period. On the *right* side of a substitution, '' is not special. If you apply this command to the line Now is the time the result will be ow is the time which is probably not what you intended.

The Backslash ''

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line

Now is the time into Now is the time? The backslash ‘’ does the job. A backslash turns off any special meaning that the next character might have; in particular, ‘’ converts the ‘’ from a ‘match anything’ into a period, so you can use it to replace the period in Now is the time like this: s//?/ The pair of characters ‘’ is considered by ed to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains PP The search /PP/ isn’t adequate, for it will find a line like THE APPLICATION OF ... because the ‘’ matches the letter ‘A’. But if you say /PP/ you will find only lines that contain ‘PP’.

The backslash can also be used to turn off special meanings for characters other than ‘’. For example, consider finding a line that contains a backslash. The search // won’t work, because the ‘’ isn’t a literal ‘’, but instead means that the second ‘/’ no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus // does work. Similarly, you can search for a forward slash ‘/’ with /// The backslash turns off the meaning of the immediately following ‘/’ so that it doesn’t terminate the /.../ construction prematurely.

As an exercise, before reading further, find two substitute commands each of which will convert the line xy into the line xy

Here are several solutions; verify that each works as advertised. s/// s/x/x/ s/y/y/

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an s command: there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like //exec //sys.fort.go // etc... you could use a colon as the delimiter _ to delete all the slashes, type s:/::g

Second, if # and @ are your character erase and line kill characters, you have to type # and @; this is true whether you’re talking to ed or any other program.

When you are adding text with a or i or c, backslash is not special, and you should only put in one backslash for each one you really want.

The Dollar Sign ‘$’

The next metacharacter, the ‘$’, stands for ‘the end of the line’. As its most obvious use, suppose you have the line Now is the and you wish to add the word ‘time’ to the end. Use the $ like this: s/$/time/ to get Now is the time Notice that a space is needed before ‘time’ in the substitute command, or you will get Now is thetime

As another example, replace the second comma in the following line with a period without altering the first: Now is the time, for all good men, The command needed is s/,$// The $ sign here provides context to make specific which comma we mean. Without it, of course, the s command would operate on the first comma to produce Now is the time for all good men,

As another example, to convert Now is the time into Now is the time? as we did earlier, we can use s/$/?/

Like ‘’, the ‘$’ has multiple meanings depending on context. In the line $s/$/$/ the first ‘$’ refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

The Circumflex ‘^’

The circumflex (or hat or caret) ‘^’ stands for the beginning of the line. For example, suppose you are looking for a line that begins with ‘the’. If you simply say /the/ you will in all likelihood find several lines that contain ‘the’ in the middle before arriving at the one you want. But with /^the/ you narrow the context, and thus arrive at the desired one more easily.

The other use of ‘^’ is of course to enable you to insert something at the beginning of a line: s/^// places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters PP you can use the command /^PP$/

The Star ‘*’

Suppose you have a line that looks like this: *text* x y *text* where *text* stands for lots of text, and there are some indeterminate number of spaces between the x and the y. Suppose the job is to replace all the spaces between x and y by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter ‘*’ comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say s/x*y/xy/ The construction ‘*’ means ‘as many spaces as possible’. Thus ‘x*y’ means ‘an x, as many spaces as possible, then a y’.

The star can be used with any character, not just space. If the original example was instead *text* x--------y *text* then all ‘−’ signs can be replaced by a single space with the command s/x-*y/xy/

Finally, suppose that the line was *text* xy *text* Can you see what trap lies in wait for the unwary? If you blindly type s/x*y/xy/ what will happen? The answer, naturally, is that it depends. If there are no other x’s or y’s on the line, then everything works, but it’s blind luck, not good management. Remember that ‘’ matches *any* single character? Then ‘*’ matches as many single characters as possible, and unless you’re careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this: *text* x *text* xy *text* y *text* then saying s/x*y/xy/ will take everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '' with '': s/x*y/xy/ Now everything works, for '*' means 'as many *periods* as possible'.

There are times when the pattern '*' is exactly what you want. For example, to change Now is the time for all good men .... into Now is the time use '*' to eat up everything after the 'for': s/for*//

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained *text* xy *text* x y *text* and we said s/x*y/xy/ the *first* 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like /x*y/ which says 'an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of '*' is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The command s/x*/y/g when applied to the line abcdef produces yaybycydyeyfy which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write s/xx*/y/g 'xx*' is one or more x's.

The Brackets '[ ]'

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like 1,$s/ˆ1*// 1,$s/ˆ2*// 1,$s/ˆ3*// and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

The construction [0123456789] matches any single digit _ the whole thing is called a 'character class'. With a character class, the job is easy. The pattern '[0123456789]*' matches zero or more digits (an entire number), so 1,$s/ˆ[0123456789]*// deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and just to confuse the issue there are essentially no special characters inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can say /[$ˆ[]/ Within [...], the '[' is not special. To get a ']' into a character class, make it the first character.

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0–9]; similarly, [a–z] stands for the lower case letters, and [A–Z] for upper case.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. This is done by beginning the class with a 'ˆ': [ˆ0-9] stands for 'any character *except* a digit'. Thus you might find the first line that doesn't begin with a tab or space by a search like /ˆ[ˆ(space)(tab)]/

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that /ˆ[ˆˆ]/ finds a line that doesn't begin with a circumflex.

The Ampersand '&'

The ampersand '&' is used primarily to save typing. Suppose you have the line Now is the time and you want to make it Now is the best time Of course you can always say s/the/the best/ but it seems silly to have to repeat the 'the'. The '&' is used to eliminate the repetition. On the *right* side of a substitute, the ampersand means 'whatever was just matched', so you can say s/the/& best/ and the '&' will stand for 'the'. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like '.*' which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length, s/*/(&)/

The ampersand can occur more than once on the right side: s/the/& best and & worst/ makes Now is the best and the worst time and s/*/&? &!!/ converts the original line into Now is the time? Now is the time!!

To get a literal ampersand, naturally the backslash is used to turn off the special meaning: s/ampersand/&/ converts the word into the symbol. Notice that '&' is not special on the left side of a substitute, only on the *right* side.

Substituting Newlines

ed provides a facility for splitting a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing (or merely because it was unwisely typed). If it looks like *text* xy *text* you can break it between the 'x' and the 'y' like this: s/xy/x y/ This is actually a single command, although it is typed on two lines. Bearing in mind that '' turns off special meanings, it seems relatively intuitive that a '' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the roff or nroff formatting command '.ul'. *text* a very big *text* The command s/very/ .ul very / converts the line into four shorter lines, preceding the word 'very' by the line '.ul', and eliminating the spaces around the 'very', all at the same time.

When a newline is substituted in, dot is left pointing at the last line created.

Joining Lines

Lines may also be joined together, but this is done with the j command instead of s. Given the lines Now is the time and supposing that dot is set to the first of them, then the command j joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a j command joins line dot to line dot+1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example, 1,$jp joins all the lines into one big one and prints it. (More on line numbers in Section 3.)

Rearranging a Line with (...)

(This section should be skipped on first reading.) Recall that '&' is a shorthand that stands for whatever was matched by the left side of an s command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form Smith, A. B. Jones, C. and so on, and you want the initials to precede the name, as in A. B. Smith C. Jones It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to 'tag' the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between (and), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol '1' refers to whatever matched the first (...) pair, '2' to the second (...), and so on.

The command 1,$s/^([^,]*),*(*)/21/ although hard to read, does the job. The first (...) matches the last name, which is any string up to the comma; this is referred to on the right side with '1'. The second (...) is whatever follows the comma and any spaces, and is referred to as '2'.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands g and v discussed in section 4 provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

3. LINE ADDRESSING IN THE EDITOR

The next general area we will discuss is that of line addressing in ed, that is, how you specify what lines are to be affected by editing commands. We have already used constructions like 1,$s/x/y/ to specify a change on all lines. And most users are long since familiar with using a single newline (or return) to print the next line, and with /thing/ to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of ?thing? to scan *backwards* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command.

Address Arithmetic

The next step is to combine the line numbers like '', '$', '/.../' and '?...?' with '+' and '–'. Thus $-1 is a command to print the next to last line of the current file (that is, one line before line '$'). For example, to recall how far you got in a previous editing session, $-5,$p prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, of course, you'll get an error message.

As another example, -3,+3p prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the '+' can be omitted: -3,3p is absolutely identical in meaning.

Another area in which you can save typing effort in specifying lines is to use '–' and '+' as line numbers by themselves. - by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines: --- moves up three lines, as does '–3'. Thus -3,+3p is also identical to the examples above.

Since '–' is shorter than '–1', constructions like -,s/bad/good/ are useful. This changes 'bad' to 'good' on the previous line and on the current line.

'+' and '–' can be used in combination with searches using '/.../' and '?...?', and with '$'. The search /thing/-- finds the line containing 'thing', and positions you two lines before it.

Repeated Searches

Suppose you ask for the search /horrible thing/ and when the line is printed you discover that it isn't the horrible thing that you wanted, so it is necessary to repeat the search again. You don't have to re-type the search, for the construction // is a shorthand for 'the previous thing that was searched for', whatever it was. This can be repeated as many times as necessary. You can also go backwards: ?? searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '//' as the left side of a substitute command, to mean 'the most recent pattern'. /horrible thing/
.... *ed prints line with 'horrible thing'* ... s//good/p To go backwards and change a line, say ??s//good/ Of course, you can still use the '&' on the right hand side of a substitute to stand for whatever got matched: //s//&&/p finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then prints the line just to verify that it worked.

Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like /thing/ you are left pointing at the next line that contains 'thing'. Then no address is required with commands like s to make a substitution on that line, or p to print it, or l to list it, or d to delete it, or a to append text after it, or c to change it, or i to insert text before it.

What happens if there was no 'thing'? Then you are left right where you were _ dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

The delete command d leaves dot pointing at the line that followed the last deleted line. When line '$' gets deleted, however, dot points at the *new* line '$'.

The line-changing commands a, c and i by default all affect the current line _ if you give no line number with them, a appends text after the current line, c changes the current line, and i inserts text before the current line.

a, c, and i behave identically in one respect _ when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say a
... text ...
... botch ... (minor error)
s/botch/correct/ (fix botched line) a
... more text ... without specifying any line number for the substitute command or for the second append command. Or you can say a
... text ...
... horrible botch ... (major error)
c (replace entire line)
... fixed up line ...

You should experiment to determine what happens if you add *no* lines with a, c or i.

The r command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say 0r to read a file in at the beginning of the text. (You can also say 0a or 1i to start adding text at the beginning.)

The w command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The w command does *not* change dot: the current line remains the same, regardless of what lines are written. This is true even if you say something like /^AB/,/^AE/w abstract which involves a context search.

Since the w command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the s command. The rule is simple _ you are left sitting on the last line that got changed. If there were no changes, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one: x1 x2 x3 Then the command -,+s/x/y/p prints the third line, which is the last one changed. But if the three lines had been x1 y2 y3 and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

Semicolon ';'

Searches with '/.../' and '?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

ab

bc

Starting at line 1, one would expect that the command /a/,/b/p prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to print a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In ed, the semicolon ';' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in our example above, the command /a/;/b/p prints the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the *second* occurrence of 'thing'. You could say /thing/ // but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to say /thing/;// This says to find the first occurrence of 'thing', set dot to that line, then find the second and print only that.

Closely related is searching for the second previous occurrence of something, as in ?something?;?? Printing the third or fourth or ... in either direction is left as an exercise.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say 1;/thing/ because this fails if 'thing' occurs on line 1. But it is possible to say 0;/thing/ (one of the few places where 0 is a legal line number), for this starts the search at line 1.

Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you hit the interrupt or delete or rubout or break key while ed is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable _ if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in some clean but unpredictable state in the middle (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if you print until you see an interesting line, then hit delete, you are *not* sitting on that line or even near it. Dot is left where it was when the p command was started.

4. GLOBAL COMMANDS

The global commands g and v are used to perform one or more editing commands on all lines that either contain g) or don't contain v) a specified pattern.

As the simplest example, the command g/UNIX/p prints all lines that contain the word 'UNIX'. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

As another example, then, g/^/p prints all the formatting commands in a file (lines that begin with '').

The v command is identical to g, except that it operates on those line that do *not* contain an occurrence of the pattern. (Don't look too hard for mnemonic significance to the letter 'v'.) So v/^/p prints all the lines that don't begin with '' _ the actual text lines.

The command that follows g or v can be anything: g/^/d deletes all lines that begin with '', and g/^$/d deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command, for this can be used to make a change and print each affected line for verification. For example, we could change the word 'Unix' to 'UNIX' everywhere, and verify that it really worked, with g/Unix/s//UNIX/gp Notice that we used '//' in the substitute command to mean 'the previous pattern', in this case, 'Unix'. The p command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a g or v to use addresses, set dot, and so on, quite freely. g/^PP/+ prints the line that follows each '.PP' command (the signal for a new paragraph in some formatting packages). Remember that '+' means 'one line past dot'. And g/topic/?^SH?1 searches for each line that contains 'topic', scans backwards until it finds a line that begins '.SH' (a section heading) and prints the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally, g/^EQ/+,/^EN/-p prints all the lines that lie between lines beginning with '.EQ' and '.EN' formatting commands.

The g and v commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain ‘thing’. Then g/thing/s/x/y/ s/a/b/ is sufficient. The ‘’ signals the g command that the set of commands continues on the next line; it terminates on the first line that does not end with ‘’. (As a minor blemish, you can’t use a substitute command to insert a newline within a g command.)

You should watch out for this problem: the command g/x/s//y/ s/a/b/ does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be ‘x’ (as expected), and sometimes it will be ‘a’ (not expected). You must spell it out, like this: g/x/s/x/y/ s/a/b/

It is also possible to execute a, c and i commands under a global command; as with other multi-line constructions, all that is needed is to add a ‘’ at the end of each line except the last. Thus to add a ‘.nf’ and ‘.sp’ command before each ‘.EQ’ line, type g/^EQ/i nf sp There is no need for a final line containing a ‘’ to terminate the i command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

5. CUT AND PASTE WITH UNIX COMMANDS

One editing area in which non-programmers seem not very confident is in what might be called ‘cut and paste’ operations _ changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

Yet most of these operations are actually quite easy, if you keep your wits about you and go cautiously. The next several sections talk about cut and paste. We will begin with the UNIX commands for moving entire files around, then discuss ed commands for operating on pieces of files.

Changing the Name of a File

You have a file named ‘memo’ and you want it to be called ‘paper’ instead. How is it done?

The UNIX program that renames files is called mv (for ‘move’); it ‘moves’ the file from one name to another, like this: mv memo paper That’s all there is to it: mv from the old name to the new name. mv oldname newname Warning: if there is already a file around with the new name, its present contents will be silently clobbered by the information from the other file. The one exception is that you can’t move a file to itself _ mv x x is illegal.

Making a Copy of a File

Sometimes what you want is a copy of a file _ an entirely fresh version. This might be because you want to work on a file, and yet save a copy in case something gets fouled up, or just because you’re paranoid.

In any case, the way to do it is with the cp command. cp stands for ‘copy’; the system is big on short command names, which are appreciated by heavy users, but sometimes a strain for novices.) Suppose you have a file called ‘good’ and you want to save a copy before you make some dramatic editing changes. Choose a name _ ‘savegood’ might be acceptable _ then type cp good savegood This copies ‘good’ onto ‘savegood’, and you now have two identical copies of the file ‘good’. (If ‘savegood’ previously contained something, it gets overwritten.)

Now if you decide at some time that you want to get back to the original state of ‘good’, you can say mv savegood good (if you’re not interested in ‘savegood’ any more), or cp savegood good if you still want to retain a safe copy.

In summary, mv just renames a file; cp makes a duplicate copy. Both of them clobber the ‘target’ file if it already exists, so you had better be sure that’s what you want to do *before* you do it.

Removing a File

If you decide you are really done with a file forever, you can remove it with the rm command: rm savegood throws away (irrevocably) the file called ‘savegood’.

Putting Two or More Files Together

The next step is the familiar one of collecting two or more files into one big one. This will be needed, for example, when the author of a paper decides that several sections need to be combined into one. There are several ways to do it, of which the cleanest, once you get used to it, is a program called cat. (Not *all* programs have two-letter names.) cat is short for ‘concatenate’, which is exactly what we want to do.

Suppose the job is to combine the files ‘file1’ and ‘file2’ into a single file called ‘bigfile’. If you say cat file the contents of ‘file’ will get printed on your terminal. If you say cat file1 file2 the contents of ‘file1’ and then the contents of ‘file2’ will *both* be printed on your terminal, in that order. So cat combines the files, all right, but it’s not much help to print them on the terminal _ we want them in ‘bigfile’.

Fortunately, there is a way. You can tell the system that instead of printing on your terminal, you want the same information put in a file. The way to do it is to add to the command line the character ≥ and the name of the file where you want the output to go. Then you can say cat file1 file2 >bigfile and the job is done. (As with cp and mv, you’re putting something into ‘bigfile’, and anything that was already there is destroyed.)

This ability to ‘capture’ the output of a program is one of the most useful aspects of the system. Fortunately it’s not limited to the cat program _ you can use it with *any* program that prints on your terminal.

We'll see some more uses for it in a moment.

Naturally, you can combine several files, not just two: cat file1 file2 file3 ... >bigfile collects a whole bunch.

Question: is there any difference between cp good savegood and cat good >savegood Answer: for most purposes, no. You might reasonably ask why there are two programs in that case, since cat is obviously all you need. The answer is that cp can do some other things as well, which you can investigate for yourself by reading the manual. For now we'll stick to simple usages.

Adding Something to the End of a File

Sometimes you want to add one file to the end of another. We have enough building blocks now that you can do it; in fact before reading further it would be valuable if you figured out how. To be specific, how would you use cp, mv and/or cat to add the file 'good1' to the end of the file 'good'?

You could try cat good good1 >temp mv temp good which is probably most direct. You should also understand why cat good good1 >good doesn't work. (Don't practice with a good 'good'!)

The easy way is to use a variant of >, called >>. In fact, >> is identical to > except that instead of clobbering the old file, it simply tacks stuff on at the end. Thus you could say cat good1 >>good and 'good1' is added to the end of 'good'. (And if 'good' didn't exist, this makes a copy of 'good1' called 'good'.)

6. CUT AND PASTE WITH THE EDITOR

Now we move on to manipulating pieces of files _ individual lines or groups of lines. This is another area where new users seem unsure of themselves.

Filenames

The first step is to ensure that you know the ed commands for reading and writing files. Of course you can't go very far without knowing r and w. Equally useful, but less well known, is the 'edit' command e. Within ed, the command e newfile says 'I want to edit a new file called *newfile,* without leaving the editor.' The e command discards whatever you're currently working on and starts over on *newfile.* It's exactly the same as if you had quit with the q command, then re-entered ed with a new file name, except that if you have a pattern remembered, then a command like // will still work.

If you enter ed with the command ed file ed remembers the name of the file, and any subsequent e, r or w commands that don't contain a filename will refer to this remembered file. Thus ed file1
... (editing) ... w (writes back in file1) e file2 (edit new file, without leaving editor)
... (editing on file2) ... w (writes back on file2)
(and so on) does a series of edits on various files without ever leaving ed and without typing the name of any file more than once. (As an aside, if you examine the sequence of commands here, you can see why many UNIX systems use e as a synonym for ed.)

You can find out the remembered file name at any time with the f command; just type f without a file name. You can also change the name of the remembered file name with f; a useful sequence is ed precious
f junk
... (editing) ... which gets a copy of a precious file, then uses f to guarantee that a careless w command won't clobber the original.

Inserting One File into Another

Suppose you have a file called 'memo', and you want the file called 'table' to be inserted just after the reference to Table 1. That is, in 'memo' somewhere is a line that says

Table 1 shows that ...

and the data contained in 'table' has to go there, probably so it will be formatted properly by nroff or troff. Now what?

This one is easy. Edit 'memo', find 'Table 1', and add the file 'table' right there: ed memo /Table 1/ *Table 1 shows that ... [response from ed]* r table The critical line is the last one. As we said earlier, the r command reads a file; here you asked for it to be read in right after line dot. An r command without any address adds lines at the end, so it is the same as $r.

Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to copy out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have TS
...[lots of stuff] TE which is the way a table is set up for the tbl program. To isolate the table in a separate file called 'table', first find the start of the table (the '.TS' line), then write out the interesting part: /^TS/ *TS [ed prints the line it found]* ,/^TE/w table and the job is done. If you are confident, you can do it all at once with /^TS/;/^TE/w table and now you have two copies, one in the file you're still editing, one in the file 'table' you've just written.

The point is that the w command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it _ don't re-type it. In the editor, say a ...lots of stuff... ...horrible line... w temp a more stuff r temp a more stuff This last example is worth studying, to be sure you appreciate what's going on.

Moving Lines Around

Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a concrete example, suppose each paragraph in the paper begins with the formatting command '.PP'. Think about it and write down the details before reading on.

The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that you are sitting on the '.PP' command that begins the paragraph, this is the sequence of commands: ,/^PP/-w temp ,//-d $r temp That is, from where you are now ('') until one line before the next 'PP' ('/^PP/–') write onto 'temp'. Then delete the same lines. Finally, read 'temp' at the end.

As we said, that's the brute force way. The easier way (often) is to use the *move* command m that ed provides _ it lets you do the whole set of operations at one crack, without any temporary file.

The m command is like many other ed commands in that it takes up to two line numbers in front that tell what lines are to be affected. It is also *followed* by a line number that tells where the lines are to go. Thus line1, line2 m line3 says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, $ signs, or other ways to specify lines.

Suppose again that you're sitting at the first line of the paragraph. Then you can say ,/^PP/-m$ That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one to after the second. Suppose that you are positioned at the first. Then m+ does it. It says to move line dot to after one line after line dot. If you are positioned on the second line, m-- does the interchange.

As you can see, the m command is more succinct and direct than writing, deleting and re-reading. When is brute force better anyway? This is a matter of personal taste _ do what you have most confidence in. The main difficulty with the m command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched m command can be a ghastly mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to issue a w command before doing anything complicated; then if you goof, it's easy to back up to where you were.

Marks

ed provides a facility for marking a line with a particular name so you can later reference it by name regardless of its actual line number. This can be handy for moving lines, and for keeping track of them even after they've been moved. The *mark* command is k; the command kx marks the current line with the name 'x'. If a line number precedes the k, that line is marked. (The mark name must be a single lower case letter.) Now you can refer to the marked line with the address ′x

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with *′a.* Then find the last line and mark it with *′b.* Now position yourself at the place where the stuff is to go and say ′a,′bm

Bear in mind that only one line can have a particular mark name associated with it at any given time.

Copying Lines

We mentioned earlier the idea of saving a line that was hard to type or used often, so as to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

ed provides another command, called t (for 'transfer') for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The t command is identical to the m command, except that instead of moving lines it simply duplicates them at the place you named. Thus 1,t duplicates the entire contents that you are editing. A more common use for t is for creating a series of lines that differ only slightly. For example, you can say a .... x .... (long line) t (make a copy) s/x/y/ (change it a bit) t (make third copy) s/y/z/ (change it a bit) and so on.

The Temporary Escape '!'

Sometimes it is convenient to be able to temporarily escape from the editor to do some other UNIX command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command ! provides a way to do this.

If you say !any UNIX command your current editing state is suspended, and the UNIX command you asked for is executed. When the command finishes, ed will signal you by printing another !; at that point you can resume editing.

You can really do *any* UNIX command, including another ed. (This is quite common, in fact.) In this case, you can even do another !.

On Berkeley UNIX systems, there is an additional (and preferable) mechanism called *job control* which lets you suspend your edit session (or, for that matter, any program), return to the shell from which you invoked that program, and issue any commands, then resume the program from the point where it was stopped. See *An Introduction to the C Shell* for more details.

7. SUPPORTING TOOLS

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how ed works, because they are all based on the editor. In this section we will give some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. More information on each can be found in [3].

Grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest, but if there are many files this can get very tedious, and if the files are really big, it may be impossible because of limits in ed.

The program grep was invented to get around these limitations. The search patterns that we have described in the paper are often called 'regular expressions', and 'grep' stands for g/re/p That describes exactly what grep does _ it prints every line in a set of files that contains a particular pattern. Thus grep ′thing′ file1 file2 file3 ... finds 'thing' wherever it occurs in any of the files 'file1', 'file2', etc. grep also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since grep and ed use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes ′...′ if it contains any non-alphabetic characters, since many such characters also mean something special to the UNIX command interpreter (the 'shell'). If you don't quote them, the command interpreter will try to interpret them before grep gets a chance.

There is also a way to find lines that *don't* contain a pattern: grep -v ′thing′ file1 file2 ... finds all lines that don't contains 'thing'. The −v must occur in the position shown. Given grep and grep −v, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y': grep x file... | grep -v y (The notation | is a 'pipe', which causes the output of the first command to be used as input to the second command; see [2].)

Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a 'script', i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every 'Unix' to 'UNIX' and every 'Gcos' to 'GCOS' in a large number of files. Then put into the file 'script' the lines g/Unix/s//UNIX/g g/Gcos/s//GCOS/g w q Now you can say ed file1 <script ed file2 <script ... This causes ed to take its commands from the prepared script. Notice that the whole job has to be planned in advance.

And of course by using the UNIX command interpreter, you can cycle through a set of files automatically, with varying degrees of ease.

Sed

sed ('stream editor') is a version of the editor with restricted capabilities but which is capable of processing unlimited amounts of input. Basically sed copies its input to its output, applying one or more editing commands to each line of input.

As an example, suppose that we want to do the 'Unix' to 'UNIX' part of the example given above, but without rewriting the files. Then the command sed ′s/Unix/UNIX/g′ file1 file2 ... applies the command 's/Unix/UNIX/g' to all lines from 'file1', 'file2', etc., and copies all lines to the output. The advantage of using sed in such a case is that it can be used with input too large for ed to handle. All the output can be collected in one place, either in a file or perhaps piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file, or on the command line, with a slightly more complex syntax. To take commands from a file, for example, sed -f cmdfile input-files...

sed has further capabilities, including conditional testing and branching, which we cannot go into here, but which are described in detail in *Sed − A Non-interactive Text Editor.*

Acknowledgement

I am grateful to Ted Dolotta for his careful reading and valuable suggestions.

References

[1] Brian W. Kernighan, *A Tutorial Introduction to the UNIX Text Editor,* Bell Laboratories internal memorandum.

[2] Brian W. Kernighan, *UNIX For Beginners,* Bell Laboratories internal memorandum.

[3] Ken L. Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories.