A tutorial for the sam command language

by Rob Pike, butchered by japanoise

sam is an interactive text editor with a command language that makes heavy use of regular expressions. Although the language is syntactically similar to ed(1), the details are interestingly different. This tutorial introduces the command language, but does not discuss the screen and mouse interface. With apologies to those unfamiliar with the Ninth Edition Blit software, it is assumed that the similarity of **sam** to mux(9), at this level makes **sam**'s mouse language easy to learn.

The **sam** command language applies identically to two environments: when running **sam** on an ordinary terminal (*via* **sam −d**), and in the command window of a *downloaded* **sam**, that is, one using the bitmap display and mouse.

Introduction

This tutorial describes the command language of **sam**, an interactive text editor that runs on Blits and some computers with bitmap displays. For most editing tasks, the mouse−based editing features are sufficient, and they are easy to use and to learn.

The command language is often useful, however, particularly when making global changes. Unlike the commands in **ed**, which are necessary to make changes, **sam** commands tend to be used only for complicated or repetitive editing tasks. It is in these more involved uses that the differences between **sam** and other text editors are most evident.

sam's language makes it easy to do some things that other editors, including programs like **sed** and **awk**, do not handle gracefully, so this tutorial serves partly as a lesson in **sam**'s manner of manipulating text. The examples below therefore concentrate entirely on the language, assuming that facility with the use of the mouse in **sam** is at worst easy to pick up. In fact, **sam** can be run without the mouse at all (not *downloaded*), by specifying the **−d** flag, and it is this domain that the tutorial occupies; the command language in these modes are identical.

A word to the Unix adept: although **sam** is syntactically very similar to **ed**, it is fundamentally and deliberately different in design and detailed semantics. You might use knowledge of **ed** to predict how the substitute command works, but you'd only be right if you had used some understanding of **sam**'s workings to influence your prediction. Be particularly careful about idioms. Idioms form in curious nooks of languages and depend on undependable peculiarities. **ed** idioms simply don't work in **sam**: **1,$s/a/b/** makes one substitution in the whole file, not one per line. **sam** has its own idioms. Much of the purpose of this tutorial is to publish them and make fluency in **sam** a matter of learning, not cunning.

The tutorial depends on familiarity with regular expressions, although some experience with a more traditional Unix editor may be helpful. To aid readers familiar with **ed**, I have pointed out in square brackets [] some of the relevant differences between **ed** and **sam**. Read these comments only if you wish to understand the differences; the lesson is about **sam**, not **sam** *vs.* **ed**. Another typographic convention is that output appears in **this font,** while typed input appears as *slanty text.*

(Note that in markdown, preformatted blocks do not support formatting, so this does not hold true for blocks of `i/o`. Also, I've occasionally used `monospace` for places where markdown couldn't handle the formatting. If in doubt, read the original (PDF) file — japanoise)

Nomenclature: **sam** keeps a copy of the text it is editing. This copy is called a *file*. To avoid confusion, I have called the permanent storage on disc a *Unix file.*

Text

To get started, we need some text to play with. Any text will do; try something from James Gosling's Emacs manual:

```
$ sam -d
a
This manual is organized in a rather haphazard manner.  The first
several sections were written hastily in an attempt to provide a
general introduction to the commands in Emacs and to try to show
the method in the madness that is the Emacs command structure.
.
```

sam −d starts **sam** running. The **a** command adds text until a line containing just a period, and sets the *current text* (also called *dot*) to what was typed — everything between the **a** and the period. [**ed** would leave dot set to only the last line.] The **p** command prints the current text:

```
p
This manual is organized in a rather haphazard manner.  The first
several sections were written hastily in an attempt to provide a
general introduction to the commands in Emacs and to try to show
the method in the madness that is the Emacs command structure.
```

[Again, **ed** would print only the last line.] The **a** command adds its text *after* dot; the **i** command is like **a,** but adds the text *before* dot.

```
i
Introduction
.
p
Introduction
```

There is also a **c** command that changes (replaces) the current text, and **d** that deletes it; these are illustrated below.

To see all the text, we can specify what text to print; for the moment, suffice it to say that `0,$` specifies the entire file. [**ed** users would probably type `1,$`, which in practice is the same thing, but see below.]

```
0,$p
Introduction
This manual is organized in a rather haphazard manner.  The first
several sections were written hastily in an attempt to provide a
general introduction to the commands in Emacs and to try to show
the method in the madness that is the Emacs command structure.
```

Except for the **w** command described below, *all* commands, including **p**, set dot to the text they touch. Thus, **a** and **i** set dot to the new text, **p** to the text printed, and so on. Similarly, all commands (except **w**) by default operate on the current text [unlike **ed**, for which some commands (such as **g**) default to the entire file].

Things are not going to get very interesting until we can set dot arbitrarily. This is done by *addresses,* which specify a piece of the file. The address **1**, for example, sets dot to the first line of the file.

```
1p
Introduction
c
Preamble
.
```

The **c** command didn't need to specify dot; the **p** left it on line one. It's therefore easy to delete the first line utterly; the last command left dot set to line one:

```
d
1p
This manual is organized in a rather haphazard manner.  The first
```

(Line numbers change to reflect changes to the file.)

The address */text/* sets dot to the first appearance of *text*, after dot. [**ed** matches the first line containing *text*.] If *text* is not found, the search restarts at the beginning of the file and continues until dot.

```
/Emacs/p
Emacs
```

It's difficult to indicate typographically, but in this example no newline appears after **Emacs**: the text to be printed is the string '**Emacs**', exactly. (The final **p** may be left off — it is the default command. When downloaded, however, the default is instead to select the text, to highlight it, and to make it visible by moving the window on the file if necessary. Thus, **/Emacs/** indicates on the display the next occurrence of the text.)

Imagine we wanted to change the word **haphazard** to **thoughtless**. Obviously, what's needed is another **c** command, but the method used so far to insert text includes a newline. The syntax for including text without newlines is to surround the text with slashes (which is the same as the syntax for text searches, but what is going on should be clear from context). The text must appear immediately after the **c** (or **a** or **i**). Given this, it is easy to make the required change:

```
/haphazard/c/thoughtless/
1p
This manual is organized in a rather thoughtless manner.  The first
```

[Changes can always be done with a **c** command, even if the text is smaller than a line]. You'll find that this way of providing text to commands is much more common than is the multiple−lines syntax. If you want to include a slash `/` in the text, just precede it with a backslash `\`, and use a backslash to protect a backslash itself.

```
/Emacs/c/Emacs\\360/
4p
general introduction to the commands in Emacs\360 and to try to show
```

We could also make this particular change by

```
/Emacs/a/\\360/
```

This is as good a place as any to introduce the **u** command, which undoes the last command. A second **u** will undo the penultimate command, and so on.

```
u
4p
general introduction to the commands in Emacs and to try to show
u
3p
```

```
This manual is organized in a rather haphazard manner.  The first
```

Undoing can only back up; there is no way to undo a previous **u**.

Addresses

We've seen the simplest forms of addresses, but there is more to learn before we can get too much further. An address selects a region in the file — a substring — and therefore must define the beginning and the end of a region. Thus, the address **13** selects from the beginning of line thirteen to the end of line thirteen, and **/Emacs/** selects from the beginning of the word '**Emacs**' to the end.

Addresses may be combined with a comma:

```
13,15
```

selects lines thirteen through fifteen. The definition of the comma operator is to select from the beginning of the left hand address (the beginning of line 13) to the end of the right hand address (the end of line 15).

A few special simple addresses come in handy: **.** (a period) represents dot, the current text, **0** (line zero) selects the null string at the beginning of the file, and **$** selects the null string at the end of the file [not the last line of the file]. Therefore,

```
0,13
```

selects from the beginning of the file to the end of line thirteen,

```
.,$
```

selects from the beginning of the current text to the end of the file, and

```
0,$
```

selects the whole file [that is, a single string containing the whole file, not a list of all the lines in the file].

These are all *absolute* addresses: they refer to specific places in the file. **sam** also has relative addresses, which depend on the value of dot, and in fact we have already seen one form: **/Emacs/** finds the first occurrence of **Emacs** searching forwards from dot. Which occurrence of **Emacs** it finds depends on the value of dot. What if you wanted the first occurrence **before** dot? Just precede the pattern with a minus sign, which reverses the direction of the search:

```
-/Emacs/
```

In fact, the complete syntax for forward searching is

```
+/Emacs/
```

but the plus sign is the default, and in practice is rarely used. Here is an example that includes it for clarity:

```
0+/Emacs/
```

selects the first occurrence of **Emacs** in the file; read it as "go to line 0, then search forwards for **Emacs**" Since the **+** is optional, this can be written **0/Emacs/**. Similarly,

```
$-/Emacs/
```

finds the last occurrence in the file, so

```
0/Emacs/,$-/Emacs/
```

selects the text from the first to last **Emacs**, inclusive. Slightly more interesting:

```
/Emacs/+/Emacs/
```

(there is an implicit **.+** at the beginning) selects the second **Emacs** following dot.

Line numbers may also be relative.

```
-2
```

selects the second previous line, and

```
+5
```

selects the fifth following line (here the plus sign is obligatory).

Since addresses may select (and dot may be) more than one line, we need a definition of 'previous' and 'following:' 'previous' means *before the beginning* of dot, and 'following' means *after the end* of dot. For example, if the file contains A*AA*A with dot set to the middle two **A**s (the slanting characters), **–/A/** sets dot to the first **A**, and **+/A/** sets dot to the last **A**. Except under odd circumstances (such as when the only occurrence of the text in the file is already the current text), the text selected by a search will be disjoint from dot.

To select the **troff –ms** paragraph containing dot, however long it is, use

```
-/.PP/,/.PP/-1
```

which will include the **.PP** that begins the paragraph, and exclude the one that ends it.

When typing relative line number addresses, the default number is **1**, so the above could be written slightly more simply:

```
-/.PP/,/.PP/-
```

What does the address **+1–1** or the equivalent **+–** mean? It looks like it does nothing, but recall that dot need not be a complete line of text. **+1** selects the line after the end of the current text, and **–1** selects the line before the beginning. Therefore **+1–1** selects the line before the line after the end of dot, that is, the complete line containing the end of dot. We can use this construction to expand a selection to include a complete line, say the first line in the file containing **Emacs**:

```
0/Emacs/+-p
general introduction to the commands in Emacs and to try to show
```

The address **+–** is an idiom.

Loops

Above, we changed one occurrence of **Emacs** to **Emacs\360**, but if the name of the editor is really changing, it would be useful to change *all* instances of the name in a single command. **sam** provides a command, **x** (extract), for just that job. The syntax is *x/pattern/command*. For each occurrence of the pattern in the selected text, **x** sets dot to the occurrence and runs command. For example, to change **Emacs** to **vi,**

```
0,$x/Emacs/c/vi/
0,$p
This manual is organized in a rather haphazard manner.  The first
several sections were written hastily in an attempt to provide a
general introduction to the commands in vi and to try to show
the method in the madness that is the vi command structure.
```

This works by subdividing the current text (**0,$** — the whole file) into appearances of its textual argument (**Emacs**), and then running the command that follows (**c/vi/**) with dot

set to the text. We can read this example as, "find all occurrences of **Emacs** in the file, and for each one, set the current text to the occurrence and run the command **c/vi/**, which will replace the current text by **vi.**" [This command is somewhat similar to **ed**'s **g** command. The differences will develop below, but note that the default address, as always, is dot rather than the whole file.]

A single **u** command is sufficient to undo an **x** command, regardless of how many individual changes the **x** makes.

```
u
0,$p
This manual is organized in a rather haphazard manner.  The first
several sections were written hastily in an attempt to provide a
general introduction to the commands in Emacs and to try to show
the method in the madness that is the Emacs command structure.
```

Of course, **c** is not the only command **x** can run. An **a** command can be used to put proprietary markings on **Emacs**:

```
0,$x/Emacs/a/{TM}/
/Emacs/+-p
general introduction to the commands in Emacs{TM} and to try to show
```

[There is no way to see the changes as they happen, as in **ed**'s **g/Emacs/s//&{TM}/p**; see the section on Multiple Changes, below.]

The **p** command is also useful when driven by an **x**, but be careful that you say what you mean;

```
0,$x/Emacs/p
EmacsEmacs
```

since **x** sets dot to the text in the slashes, printing only that text is not going to be very informative. But the command that **x** runs can contain addresses. For example, if we want to print all lines containing **Emacs**, just use **+−**:

```
0,$x/Emacs/+-p
general introduction to the commands in Emacs{TM} and to try to show
the method in the madness that is the Emacs{TM} command structure.
```

Finally, let's restore the state of the file with another **x** command, and make use of a handy shorthand: a comma in an address has its left side default to **0**, and its right side default to **$**, so the easy−to−type address **,** refers to the whole file:

```
,x/Emacs/ /{TM}/d
,p
This manual is organized in a rather haphazard manner.  The first
several sections were written hastily in an attempt to provide a
general introduction to the commands in Emacs and to try to show
the method in the madness that is the Emacs command structure.
```

Notice what this **x** does: for each occurrence of Emacs, find the **{TM}** that follows, and delete it.

The 'text' **sam** accepts for searches in addresses and in **x** commands is not simple text, but rather *regular expressions*. Unix has several distinct interpretations of regular expressions. The form used by **sam** is that of regexp (6), including parentheses **()** for grouping and an 'or' operator **|** for matching strings in parallel. **sam** also matches the character sequence **\n** with a newline character. Replacement text, such as used in the **a** and **c** commands, is

still plain text, but the sequence **\n** represents newline in that context, too.

Here is an example. Say we wanted to double space the document, that is, turn every newline into two newlines. The following all do the job:

```
,x/\n/ a/\n/
,x/\n/ c/\n\n/
,x/$/ a/\n/
,x/^/ i/\n/
```

The last example is slightly different, because it puts a newline *before* each line; the other examples place it after. The first two examples manipulate newlines directly [something outside **ed**'s ken]; the last two use regular expressions: **$** is the empty string at the end of a line, while **^** is the empty string at the beginning.

These solutions all have a possible drawback: if there is already a blank line (that is, two consecutive newlines), they make it much larger (four consecutive newlines). A better method is to extend every group of newlines by one:

```
,x/\n+/ a/\n/
```

The regular expression operator **+** means 'one or more;' **\n+** is identical to `\n\n*`. Thus, this example takes every sequence of newlines and adds another to the end.

A more common example is indenting a block of text by a tab stop. The following all work, although the first is arguably the cleanest (the blank text in slashes is a tab):

```
,x/^/a/     /
,x/^/c/     /
,x/.*\n/i/     /
```

The last example uses the pattern (idiom, really) `.*\n` to match lines: `.*` matches the longest possible string of non–newline characters. Taking initial tabs away is just as easy:

```
,x/^     /d
```

In these examples I have specified an address (the whole file), but in practice commands like these are more likely to be run without an address, using the value of dot set by selecting text with the mouse.

Conditionals

The **x** command is a looping construct: for each match of a regular expression, it extracts (sets dot to) the match and runs a command. **sam** also has a conditional, **g**: *g/pattern/command* runs the command if dot contains a match of the pattern *without changing the value of dot*. The inverse, **v**, runs the command if dot does *not* contain a match of the pattern. (The letters **g** and **v** are historical and have no mnemonic significance. You might think of **g** as 'guard.') [**ed** users should read the above definitions very carefully; the **g** command in **sam** is fundamentally different from that in **ed**.] Here is an example of the difference between **x** and **g**:

```
,x/Emacs/c/vi/
```

changes each occurrence of the word **Emacs** in the file to the word **vi**, but

```
,g/Emacs/c/vi/
```

changes the *whole file* to **vi** if there is the word **Emacs** anywhere in the file.

Neither of these commands is particularly interesting in isolation, but they are valuable when combined with **x** and with themselves.

Composition

One way to think about the **x** command is that, given a selection (a value of dot) it iterates through interesting subselections (values of dot within). In other words, it takes a piece of text and cuts it into smaller pieces. But the text that it cuts up may already be a piece cut by a previous **x** command or selected by a **g**. **sam**'s most interesting property is the ability to define a sequence of commands to perform a particular task. †(The obvious analogy with shell pipelines is only partially valid, because the individual **sam** commands are all working on the same text; it is only how the text is sliced up that is changing.)

A simple example is to change all occurrences of **Emacs** to **emacs**; certainly the command

```
,x/Emacs/ c/emacs/
```

will work, but we can use an **x** command to save retyping most of the word **Emacs**:

```
,x/Emacs/ x/E/ c/e/
```

(Blanks can be used to separate commands on a line to make them easier to read.) What this command does is find all occurrences of **Emacs (,x/Emacs/),** and then *with dot set to that text,* find all occurrences of the letter **E** (**x/E/**), and then *with dot set to that text,* run the command **c/e/** to change the character to lower case. Note that the address for the command — the whole file, specified by a comma — is only given to the leftmost piece of the command; the rest of the pieces have dot set for them by the execution of the pieces to their left.

As another simple example, consider a problem solved above: printing all lines in the file containing the word **Emacs**:

```
,x/.*\n/ g/Emacs/p
general introduction to the commands in Emacs and to try to show
the method in the madness that is the Emacs command structure.
```

This command says to break the file into lines (`,x/.*\n/`), and for each line that contains the string **Emacs** (**g/Emacs/**), run the command **p** with dot set to the line (not the match of **Emacs**), which prints the line. To save typing, because `.*\n` is a common pattern in **x** commands, if the **x** is followed immediately by a space, the pattern `.*\n` is assumed. Therefore, the above could be written more succinctly:

```
,x g/Emacs/p
```

The solution we used before was

```
,x/Emacs/+-p
```

which runs the command **+-p** with dot set to each match of **Emacs** in the file (recall that the idiom **+-p** prints the line containing the end of dot).

The two commands usually produce the same result (the **+-p** form will print a line twice if it contains **Emacs** twice). Which is better? **,x/Emacs/+-p** is easier to type and will be much faster if the file is large and there are few occurrences of the string, but it is really an odd special case. `,x/.*\n/ g/Emacs/p` is slower — it breaks each line out separately, then examines it for a match — but is conceptually cleaner, and generalizes more easily. For example, consider the following piece of the Emacs manual:

```
command name="append-to-file", key="[unbound]"
Takes the contents of the current buffer and appends it to the
named file. If the file doesn't exist, it will be created.
```

```
command name="apropos", key="ESC-?"
Prompts for a keyword and then prints a list of those commands
whose short description contains that keyword.  For example,
if you forget which commands deal with windows, just type
"@b[ESC-?]@t[window]@b[ESC]".

...and so on...
```

This text consists of groups of non–empty lines, with a simple format for the text within each group. Imagine that we wanted to find the description of the 'apropos' command. The problem is to break the file into individual descriptions, and then to find the description of 'apropos' and to print it. The solution is straightforward:

```
,x/(.+\n)+/\ g/command\ name="apropos"/p
command name="apropos", key="ESC-?"
Prompts for a keyword and then prints a list of those commands
whose short description contains that keyword.  For example,
if you forget which commands deal with windows, just type
"@b[ESC-?]@t[window]@b[ESC]".
```

The regular expression **(.+\n)+** matches one or more lines with one or more characters each, that is, the text between blank lines, so **,x/(.+\n)+/** extracts each description; then **g/command name="apropos"/** selects the description for 'apropos' and **p** prints it.

Imagine that we had a C program containing the variable **n**, but we wanted to change it to **num**. This command is a first cut:

```
,x/n/ c/num/
```

but is obviously flawed: it will change all **n**'s in the file, not just the *identifier* **n**. A better solution is to use an **x** command to extract the identifiers, and then use **g** to find the **n**'s:

```
,x/[a-zA-Z_][a-zA-Z_0-9]*/ g/n/ v/../ c/num/
```

It looks awful, but it's fairly easy to understand when read left to right. A C identifier is an alphabetic or underscore followed by zero or more alphanumerics or underscores, that is, matches of the regular expression `[a-zA-Z_][a-zA-Z_0-9]*`. The **g** command selects those identifiers containing **n**, and the **v** is a trick: it rejects those identifiers containing more than one character. Hence the **c/num/** applies only to free–standing **n**'s.

There is still a problem here: we don't want to change **n**'s that are part of the character constant **\n**. There is a command **y**, complementary to **x**, that is just what we need: *y/pattern/command* runs the command on the pieces of text *between* matches of the pattern; if **x** selects, **y** rejects. Here is the final command:

```
,y/\\n/ x/[a-zA-Z_][a-zA-Z_0-9]*/ g/n/ v/../ c/num/
```

The **y/\\n/** (with backslash doubled to make it a literal character) removes the two–character sequence **\n** from consideration, so the rest of the command will not touch it. There is more we could do here; for example, another **y** could be prefixed to protect comments in the code. I won't elaborate the example any further, but you should have an idea of the way in which the looping and conditional commands in **sam** may be composed to do interesting things.

Grouping

There is another way to arrange commands. By enclosing them in brace brackets {}, commands may be applied in parallel. This example uses the **=** command, which reports the line and character numbers of dot, together with **p**, to report on appearances of **Emacs** in

our original file:

```
,p
This manual is organized in a rather haphazard manner. The first
several sections were written hastily in an attempt to provide a
general introduction to the commands in Emacs and to try to show
the method in the madness that is the Emacs command structure.
,x/Emacs/{
    =
    +-p
}
3; #171,#176
general introduction to the commands in Emacs and to try to show
4; #234,#239
the method in the madness that is the Emacs command structure.
```

(The number before the semicolon is the line number; the numbers beginning with **#** are character numbers.) As a more interesting example, consider changing all occurrences of **Emacs** to **vi** and vice versa. We can type

```
,x/Emacs|vi/{
    g/Emacs/ c/vi/
    g/vi/ c/Emacs/
}
```

or even

```
,x/[a-zA-Z]+/{
    g/Emacs/ v/...../ c/vi/
    g/vi/ v/.../ c/Emacs/
}
```

to make sure we don't change strings embedded in words.

Multiple Changes

You might wonder why, once **Emacs** has been changed to **vi** in the above example, the second command in the braces doesn't put it back again. The reason is that the commands are run in parallel: within any top–level **sam** command, all changes to the file refer to the state of the file before any of the changes in that command are made. After all the changes have been determined, they are all applied simultaneously.

This means, as mentioned, that commands within a compound command see the state of the file before any of the changes apply. This method of evaluation makes some things easier (such as the exchange of **Emacs** and **vi**), and some things harder. For instance, it is impossible to use a **p** command to print the changes as they happen, because they haven't happened when the **p** is executed. An indirect ramification is that changes must occur in forward order through the file, and must not overlap.

Unix

sam has a few commands to connect to Unix processes. The simplest is **!**, which runs the command with input and output connected to the terminal.

```
!date
Wed May 28 23:25:21 EDT 1986
!
```

(When downloaded, the input is connected to **/dev/null** and only the first few lines of output are printed; any overflow is stored in **$HOME/sam.err**.) The final **!** is a prompt to indicate when the command completes.

Slightly more interesting is **>**, which provides the current text as standard input to the Unix command:

```
1,2 >wc
      2       22      131
!
```

The complement of **>** is, naturally, **<**: it replaces the current text with the standard output of the Unix command:

```
1 <date
!
1p
Wed May 28 23:26:44 EDT 1986
```

The last command is **|**, which is a combination of **<** and **>**: the current text is provided as standard input to the Unix command, and the Unix command's standard output is collected and used to replace the original text. For example,

```
,| sort
```

runs sort(1) on the file, sorting the lines of the text lexicographically. Note that **<**, **>** and **|** are **sam** commands, not Unix shell operators.

The next example converts all appearances of **Emacs** to upper case using tr(1):

```
,x/Emacs/ | tr a-z A-Z
```

tr is run once for each occurrence of **Emacs**. Of course, you could do this example more efficiently with a simple **c** command, but here's a trickier one: given a Unix mail box as input, convert all the **Subject** headers to distinct fortunes:

```
,x/^Subject:.*\n/ x/[^:]*\n/ < /usr/games/fortune
```

(The regular expression `[^:]` refers to any character *except* **:** and newline; the negation operator **^** excludes newline from the list of characters.) Again, **/usr/games/fortune** is run once for each **Subject** line, so each **Subject** line is changed to a different fortune.

A few other text commands

For completeness, I should mention three other commands that manipulate text. The **m** command moves the current text to after the text specified by the (obligatory) address after the command. Thus

```
/Emacs/+- m 0
```

moves the next line containing **Emacs** to the beginning of the file. Similarly, **t** (another historic character) copies the text:

```
/Emacs/+- t 0
```

would make, at the beginning of the file, a copy of the next line containing **Emacs**.

The third command is more interesting: it makes substitutions. Its syntax is *s/pattern/replacement/*. Within the current text, it finds the first occurrence of the pattern and replaces it by the replacement text, leaving dot set to the entire address of the substitution.

```
1p
This manual is organized in a rather haphazard manner.  The first
```

```
s/haphazard/thoughtless/
p
This manual is organized in a rather thoughtless manner.  The first
```

Occurrences of the character **&** in the replacement text stand for the text matching the pattern.

```
s/T/"&&&&"/
p
"TTTT"his manual is organized in a rather thoughtless manner.  The first
```

There are two variants. The first is that a number may be specified after the **s**, to indicate which occurrence of the pattern to substitute; the default is the first.

```
s2/is/was/
p
"TTTT"his manual was organized in a rather thoughtless manner.  The first
```

The second is that suffixing a **g** (global) causes replacement of all occurrences, not just the first.

```
s/[a-zA-Z]/x/g
p
"xxxx"xxx xxxxxx xxx xxxxxxxxx xx x xxxxxx xxxxxxxxxx xxxxxxx  xxx xxxxx
```

Notice that in all these examples dot is left set to the entire line.

[The substitute command is vital to **ed**, because it is the only way to make changes within a line. It is less valuable in **sam** , in which the concept of a line is much less important. For example, many **ed** substitution idioms are handled well by **sam**'s basic commands. Consider the commands

```
s/good/bad/
s/good//
s/good/& bye/
```

which are equivalent in **sam** to

```
/good/c/bad/
/good/d
/good/a/ bye/
```

and for which the context search is likely unnecessary because the desired text is already dot. Also, beware this **ed** idiom:

```
1,$s/good/bad/
```

which changes the first **good** on each line; the same command in **sam** will only change the first one in the whole file. The correct **sam** version is

```
,x s/good/bad/
```

but what is more likely meant is

```
,x/good/ c/bad/
```

sam operates under different rules.]

Files

So far, we have only been working with a single file, but **sam** is a multi−file editor. Only one file may be edited at a time, but it is easy to change which file is the 'current' file for editing. To see how to do this, we need a **sam** with a few files; the easiest way to do this

is to start it with a list of Unix file names to edit.

$ echo *.ms conquest.ms death.ms emacs.ms famine.ms slaughter.ms $ sam −d* .ms −. conquest.ms

(I'm sorry the Horsemen don't appear in liturgical order.) The line printed by **sam** is an indication that the Unix file **conquest.ms** has been read, and is now the current file. **sam** does not read the Unix file until the associated **sam** file becomes current.

The **n** command prints the names of all the files:

```
n
 −. conquest.ms
 −  death.ms
 −  emacs.ms
 −  famine.ms
 −  slaughter.ms
```

This list is also available in the menu on mouse button 3. The command **f** tells the name of just the current file:

```
f
 −. conquest.ms
```

The characters to the left of the file name encode helpful information about the file. The minus sign becomes a plus sign if the file has a window open, and an asterisk if more than one is open. The period (another meaning of dot) identifies the current file. The leading blank changes to an apostrophe if the file is different from the contents of the associated Unix file, as far as **sam** knows. This becomes evident if we make a change.

```
1d
f
 '−. conquest.ms
```

If the file is restored by an undo command, the apostrophe disappears.

```
u
f
 −. conquest.ms
```

The file name may be changed by providing a new name with the **f** command:

```
f pestilence.ms
'−. pestilence.ms
```

f prints the new status of the file, that is, it changes the name if one is provided, and prints the name regardless. A file name change may also be undone.

```
u
f
 −. conquest.ms
```

When **sam** is downloaded, the current file may be changed simply by selecting the desired file from the menu (selecting the same file subsequently cycles through the windows opened on the file). Otherwise, the **b** command can be used to choose the desired file: †(A bug prevents the **b** command from working when downloaded. Because the menu is more convenient anyway, and because the method of choosing files from the command language is slated to change, the bug hasn't been fixed.)

```
b emacs.ms
 −. emacs.ms
```

Again, **sam** prints the name (actually, executes an implicit **f** command) because the Unix file **emacs.ms** is being read for the first time. It is an error to ask for a file **sam** doesn't know about, but the **B** command will prime **sam**'s menu with a new file, and make it current.

```
b flood.pic
?no such file `flood.pic'
B flood.pic
 -. flood.pic
n
 -  conquest.ms
 -  death.ms
 -  emacs.ms
 -  famine.ms
 -. flood.pic
 -  slaughter.ms
```

Both **b** and **B** will accept a list of file names. **b** simply takes the first file in the list, but **B** loads them all. The list may be typed on one line —

```
B devil.tex satan.tex 666.tex emacs.tex
```

— or generated by a Unix command —

```
B <echo *.tex
```

The latter form requires a Unix command; **sam** does not understand the shell file name metacharacters, so `B *.tex` attempts to load a single file named `*.tex`. (The **<** form is of course derived from **sam**'s **<** command.) **echo** is not the only useful command to run subservient to **B**; for example,

```
B <grep -l Emacs *
```

will load only those files containing the string **Emacs**. Finally, a special case: a **B** with no arguments creates an empty, nameless file within **sam**.

The complement of **B** is **D**:

```
D devil.tex satan.tex 666.tex emacs.tex
```

eradicates the files from **sam**'s memory (not from the Unix machine's disc). **D** without any file names removes the current file from **sam**.

There are three other commands that relate the current file to Unix files. The **w** command writes the file to disc; without arguments, it writes the entire file to the Unix file associated with the current file in **sam** (it is the only command whose default address is not dot). Of course, you can specify an address to be written, and a different file name, with the obvious syntax:

```
1,2w /tmp/revelations
/tmp/revelations: #44
```

sam responds with the file name and the number of characters written to the file. The **write** command on the button 3 menu is identical in function to an unadorned **w** command.

The other two commands, **e** and **r**, read data from Unix files. The **e** command clears out the current file, reads the data from the named file (or uses the current file's old name if none is explicitly provided), and sets the file name. It's much like a **B** command, but puts the information in the current file instead of a new one. **e** without any file name is

therefore an easy way to refresh **sam**'s copy of a Unix file. [Unlike in **ed**, **e** doesn't complain if the file is modified. The principle is not to protect against things that can be undone if wrong.] Since its job is to replace the whole text, **e** never takes an address.

The **r** command is like **e**, but it doesn't clear the file: the text in the Unix file replaces dot, or the specified text if an address is given.

```
r emacs.ms
```

has essentially the effect of

```
<cat emacs.ms
```

The commands **r** and **w** will set the name of the file if the current file has no name already defined; **e** sets the name even if the file already has one.

There is a command, analogous to **x**, that iterates over files instead of pieces of text: **X** (capital **x**). The syntax is easy; it's just like that of **x** — *X/pattern/command*. (The complementary command is **Y**, analogous to **y**.) The effect is to run the command in each file whose menu entry (that is, whose line printed by an **f** command) matches the pattern. For example, since an apostrophe identifies modified files,

```
X/'/ w
```

writes the changed files out to disc. Here is a longer example: find all uses of a particular variable in the C source files:

```
X/\.c$/ ,x/variable/+-p
```

We can use an **f** command to identify which file the variable appears in:

```
X/\.c$/ ,g/variable/ {
    f
    ,x/variable/+-{
        =
        p
    }
}
```

Here, the **g** command guarantees that only the names of files containing the variable will be printed (but beware that **sam** may confuse matters by printing the names of files it reads in during the command). The **=** command shows where in the file the variable appears, and the **p** command prints the line.

The **D** command is handy as the target of an **X**. This example deletes from the menu all C files that do not contain a particular variable:

```
X/\.c$/ ,v/variable/ D
```

If no pattern is provided for the **X**, the command (which defaults to **f**) is run in all files, so

```
X D
```

cleans **sam** up for a fresh start.

But rather than working any further, let's stop now:

```
q
$
```

Some of the file manipulating commands can be undone: undoing a **f**, **e**, or **r** restores the previous state of the file, but **w**, **B** and **D** are irrevocable. And, of course, so is **q**.