Editing files via scripts with ed

Why ed?

Like `sed`, `ed` is a line editor. However, if you try to change file contents with `sed`, and the file is open elsewhere and read by some process, you will find out that GNU `sed` and its `-i` option will not allow you to edit the file. There are circumstances where you may need that, e.g. editing active and open files, the lack of GNU, or other `sed`, with "in-place" option available.

Why `ed`?

- maybe your `sed` doesn't support in-place edit
- maybe you need to be as portable as possible
- maybe you need to really edit in-file (and not create a new file like GNU `sed`)
- last but not least: standard `ed` has very good editing and addressing possibilities, compared to standard `sed`

Don't get me wrong, this is **not** meant as anti-`sed` article! It's just meant to show you another way to do the job.

Commanding ed

Since `ed` is an interactive text editor, it reads and executes commands that come from `stdin`. There are several ways to feed our commands to ed:

Pipelines

```
echo '<ED-COMMANDS>' | ed <FILE>
```

To inject the needed newlines, etc. it may be easier to use the builtin command, `printf` ("help printf"). Shown here as an example Bash function to prefix text to file content:

```
# insertHead "$text" "$file"

insertHead() {
  printf '%s\n' H 1i "$1" . w | ed -s "$2"
}
```

Here-strings

```
ed <FILE> <<< '<ED-COMMANDS>'
```

Here-documents

```
ed <FILE> <<EOF
<ED-COMMANDS>
EOF
```

Which one you prefer is your choice. I will use the here-strings, since it looks best here IMHO ().

There are other ways to provide input to `ed`. For example, process substitution. But these should be enough for daily needs.

Since `ed` wants commands separated by newlines, I'll use a special Bash quoting method, the C-like strings `$'TEXT'`, as it can interpret a set of various escape sequences and special characters. I'll use the `-s` option to make it less verbose.

The basic interface

Check the `ed` manpage for details

Similar to `vi` or `vim`, `ed` has a "command mode" and an "interactive mode". For non-interactive use, the command mode is the usual choice.

Commands to `ed` have a simple and regular structure: zero, one, or two addresses followed by a single-character command, possibly followed by parameters to that command. These addresses specify one or more lines in the text buffer. Every command that requires addresses has default addresses, so the addresses can often be omitted.

The line addressing is relative to the *current line*. If the edit buffer is not empty, the initial value for the *current line* shall be the last line in the edit buffer, otherwise zero. Generally, the *current line* is the last line affected by a command. All addresses can only address single lines, not blocks of lines!

Line addresses or commands using *regular expressions* interpret POSIX Basic Regular Expressions (BRE). A null BRE is used to reference the most recently used BRE. Since `ed` addressing is only for single lines, no RE can ever match a newline.

Debugging your ed scripts

By default, `ed` is not very talkative and will simply print a "?" when an error occurs. Interactively you can use the `h` command to get a short message explaining the last error. You can also turn on a mode that makes `ed` automatically print this message with the `H` command. It is a good idea to always add this command at the beginning of your ed scripts:

```
bash > ed -s file <<< $'H\n,df'
?
script, line 2: Invalid command suffix
```

While working on your script, you might make errors and destroy your file, you might be tempted to try your script doing something like:

```
# Works, but there is better

# copy my original file
cp file file.test

# try my script on the file
ed -s file.test <<< $'H\n<ed commands>\nw'

# see the results
cat file.test
```

There is a much better way though, you can use the ed command `p` to print the file, now your testing would look like:

```
ed -s file <<< $'H\n<ed commands>\n,p'
```

the `,` (comma) in front of the `p` command is a shortcut for `1,$` which defines an address range for the first to the last line, `,p` thus means print the whole file, after it has been modified. When your script runs sucessfully, you only have to replace the `,p` by a `w`.

Of course, even if the file is not modified by the `p` command, **it's always a good idea to have a backup copy!**

Editing your files

Most of these things can be done with `sed`. But there are also things that can't be done in `sed` or can only be done with very complex code.

Simple word substitutions

Like `sed`, `ed` also knows the common `s/FROM/TO/` command, and it can also take line-addresses. **If no substitution is made on the addressed lines, it's considered an error.**

Substitutions through the whole file

```
ed -s test.txt <<< $',s/Windows(R)-compatible/POSIX-conform/g\nw'
```

Note: The comma as single address operator is an alias for `1,$` ("all lines").

Substitutions in specific lines

On a line containing `fruits`, do the substitution:

```
ed -s test.txt <<< $'/fruits/s/apple/banana/g\nw'
```

On the 5th line after the line containing `fruits`, do the substitution:

```
ed -s test.txt <<< $'/fruits/+5s/apple/banana/g\nw'
```

Block operations

Delete a block of text

The simple one is a well-known (by position) block of text:

```
# delete lines number 2 to 4 (2, 3, 4)
ed -s test.txt <<< $'2,5d\nw'
```

This deletes all lines matching a specific regular expression:

```
# delete all lines matching foobar
ed -s test.txt <<< $'g/foobar/d\nw'
```

g/regexp/ applies the command following it to all the lines matching the regexp

Move a block of text

…using the `m` command: `<ADDRESS> m <TARGET-ADDRESS>`

This is definitely something that can't be done easily with sed.

```
# moving lines 5-9 to the end of the file
ed -s test.txt <<< $'5,9m$\nw'

# moving lines 5-9 to line 3
ed -s test.txt <<< $'5,9m3\nw'
```

Copy a block of text

…using the `t` command: `<ADDRESS> t <TARGET-ADDRESS>`

You use the `t` command just like you use the `m` (move) command.

```
# make a copy of lines 5-9 and place it at the end of the file
ed -s test.txt <<< $'5,9t$\nw'

# make a copy of lines 5-9 and place it at line 3
ed -s test.txt <<< $'5,9t3\nw'
```

Join all lines

…but leave the final newline intact. This is done by an extra command: `j` (join).

```
ed -s file <<< $'1,$j\nw'
```

Compared with two other methods (using `tr` or `sed`), you don't have to delete all newlines and manually add one at the end.

File operations

Insert another file

How do you insert another file? As with `sed`, you use the `r` (read) command. That inserts another file at the line before the last line (and prints the result to stdout - `,p`):

```
ed -s FILE1 <<< $'$-1 r FILE2\n,p'
```

To compare, here's a possible `sed` solution which must use Bash arithmetic and the external program `wc`:

```
sed "$(($(wc -l < FILE1)-1))r FILE2" FILE1

# UPDATE here's one which uses GNU sed's "e" parameter for the s-comm
and
#   it executes the commands found in pattern space. I'll take that a
s a
#   security risk, but well, sometimes GNU > security, you know...
sed '${h;s/.*/cat FILE2/e;G}' FILE1
```

Another approach, in two invocations of sed, that avoids the use of external commands completely:

```
sed $'${s/$/\\n-||-/;r FILE2\n}' FILE1 | sed '0,/-||-/{//!h;N;//
D};$G'
```

Pitfalls

ed is not sed

ed and sed might look similar, but the same command(s) might act differently:

/foo/d

In sed /foo/d will delete all lines matching foo, in ed the commands are not repeated on each line so this command will search the next line matching foo and delete it. If you want to delete all lines matching foo, or do a subsitution on all lines matching foo you have to tell ed about it with the g (global) command:

```
echo $'1\n1\n3' > file

#replace all lines matching 1 by "replacement"
ed -s file <<< $'g/1/s/1/replacement/\n,p'

#replace the first line matching 1 by "replacement"
#(because it starts searching from the last line)
ed -s file <<< $'s/1/replacement/\n,p'
```

an error stops the script

You might think that it's not a problem and that the same thing happens with sed and you're right, with the exception that if ed does not find a pattern it's an error, while sed just continues with the next line. For instance, let's say that you want to change foo to bar on

the first line of the file and add something after the next line, ed will stop if it cannot find foo on the first line, sed will continue.

```
#Gnu sed version
sed -e '1s/foo/bar/' -e '$a\something' file

#First ed version, does nothing if foo is not found on the first lin
e:
ed -s file <<< $'H\n1s/foo/bar/\na\nsomething\n.\nw'
```

If you want the same behaviour you can use g/foo/ to trick ed. g/foo/ will apply the command on all lines matching foo, thus the substitution will succeed and ed will not produce an error when foo is not found:

```
#Second version will add the line with "something" even if foo is not
found
ed -s file <<< $'H\n1g/foo/s/foo/bar/\na\nsomething\n.\nw'
```

In fact, even a substitution that fails after a g/ / command does not seem to cause an error, i.e. you can use a trick like g/./s/foo/bar/ to attempt the substitution on all non blank lines

here documents

shell parameters are expanded

If you don't quote the delimiter, $ has a special meaning. This sounds obvious but it's easy to forget this fact when you use addresses like $-1 or commands like $a. Either quote the $ or the delimiter:

```
#fails
ed -s file << EOF
$a
last line
.
w
EOF

#ok
ed -s file << EOF
\$a
last line
.
w
EOF

#ok again
ed -s file << 'EOF'
$a
last line
.
w
EOF
```

"." is not a command

The . used to terminate the command "a" must be the only thing on the line. take care if you indent the commands:

```
#ed doesn't care about the spaces before the commands, but the . must
be the only thing on the line:
ed -s file << EOF
    a
my content
.
    w
EOF
```

Simulate other commands

Keep in mind that in all the examples below, the entire file will be read into memory.

A simple grep

```
ed -s file <<< 'g/foo/p'

# equivalent
ed -s file <<< 'g/foo/'
```

The name `grep` is derived from the notaion `g/RE/p` (global ⇒ regular expression ⇒ print). ref http://www.catb.org/~esr/jargon/html/G/grep.html (http://www.catb.org/~esr/jargon/html/G/grep.html)

wc -l

Since the default for the `ed` "print line number" command is the last line, a simple `=` (equal sign) will print this line number and thus the number of lines of the file:

```
ed -s file <<< '='
```

cat

Yea, it's a joke…

```
ed -s file <<< $',p'
```

…but a similar thing to `cat` showing line-endings and escapes can be done with the `list` command (l):

```
ed -s file <<< $',l'
```

Fix Me! to be continued

Links

Reference:

- Gnu ed (http://www.gnu.org/software/ed/manual/ed_manual.html) - if we had to guess, you're probably using this one.
- POSIX ed (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/ed.html#tag_20_38), ex (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/ex.html#tag_20_40), and vi (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/vi.html#tag_20_152)
- http://sdf.lonestar.org/index.cgi?tutorials/ed (http://sdf.lonestar.org/index.cgi?tutorials/ed) - ed cheatsheet on sdf.org

Misc info / tutorials:

- How can I replace a string with another string in a variable, a stream, a file, or in all the files in a directory? (http://mywiki.wooledge.org/BashFAQ/021) - BashFAQ
- http://wolfram.schneider.org/bsd/7thEdManVol2/edtut/edtut.pdf (http://wolfram.schneider.org/bsd/7thEdManVol2/edtut/edtut.pdf) - Old but still relevant ed tutorial.

Discussion

noname, 2010/04/12 05:45 ()

wc -l Since the current line (the line addressed when no explicit address is given) is the last line after startup, a = (equal sign) there will print the line number of the last line, i.e. the number of lines of the file: the = (equal sign) doesn't depend where we are now, by default it always prints the number of $ line try **ed -s /etc/passwd «< $'1\n='**

Jan Schampera, 2010/04/12 14:19 ()

Aaaah you're absolutely right. My `ed` manpage mentions "default addresses". Thanks!

Steve Terpe (https://github.com/sterpe), 2014/12/02 16:42 ()

Really crazy, use `sed` to `ed` multiple files in place:

```
ls *.js | sed "s/.*/ed & << EOF\\
 0a\\
 \\/** @jsx React.DOM *\\/\\
 .\\
 wq\\
 EOF/" | sh
```

foo, 2015/05/25 07:18 ()

delete lines number 2 to 4 (2, 3, 4)

Comment or code should be changed.

~ $ cat » f

111

222

333

444

555

666

~ $ ed -s f «< $'2,5d\nw'

~ $ cat f

111

666

howto/edit-ed.txt Last modified: 2015/08/08 20:00 by bill_thomson