# Common threads: Sed by example, Part 3

## Taking it to the next level: Data crunching, sed style

Daniel Robbins

November 01, 2000

In this conclusion of the sed series, Daniel Robbins gives you a true taste of the power of sed. After introducing a handful of essential sed scripts, he'll demonstrate some radical sed scripting by converting a Quicken .QIF file into a text-readable format. This conversion script is not only functional, it also serves as an excellent example of sed scripting power.

## Muscular sed

In my second sed article, I offered examples that demonstrated how sed works, but very few of these examples actually did anything particularly *useful*. In this final sed article, it's time to change that pattern and put sed to good use. I'll show you several excellent examples that not only demonstrate the power of sed, but also do some really neat (and handy) things. For example, in the second half of the article, I'll show you how I designed a sed script that converts a .QIF file from Intuit's Quicken financial program into a nicely formatted text file. Before doing that, we'll take a look at some less complicated yet useful sed scripts.

## Text translation

Our first practical script converts UNIX-style text to DOS/Windows format. As you probably know, DOS/Windows-based text files have a CR (carriage return) and LF (line feed) at the end of each line, while UNIX text has only a line feed. There may be times when you need to move some UNIX text to a Windows system, and this script will perform the necessary format conversion for you.

```
$ sed -e 's/$/\r/' myunix.txt > mydos.txt
```

In this script, the '$' regular expression will match the end of the line, and the '\r' tells sed to insert a carriage return right before it. Insert a carriage return before a line feed, and presto, a CR/LF ends each line. Please note that the '\r' will be replaced with a CR only when using GNU sed 3.02.80 or later. If you haven't installed GNU sed 3.02.80 yet, see my first sed article for instructions on how to do this.

I can't tell you how many times I've downloaded some example script or C code, only to find that it's in DOS/Windows format. While many programs don't mind DOS/Windows format CR/LF text

files, several programs definitely do -- the most notable being bash, which chokes as soon as it encounters a carriage return. The following sed invocation will convert DOS/Windows format text to trusty UNIX format:

```
$ sed -e 's/.$//' mydos.txt > myunix.txt
```

The way this script works is simple: our substitution regular expression matches the last character on the line, which happens to be a carriage return. We replace it with nothing, causing it to be deleted from the output entirely. If you use this script and notice that the last character of every line of the output has been deleted, you've specified a text file that's already in UNIX format. No need for that!

## Reversing lines

Here's another handy little script. This one will reverse lines in a file, similar to the "tac" command that's included with most Linux distributions. The name "tac" may be a bit misleading, because "tac" doesn't reverse the position of characters on the line (left and right), but rather the position of lines in the file (up and down). Tacing the following file:

```
foo

bar

oni
```

....produces the following output:

```
oni

bar

foo
```

We can do the same thing with the following sed script:

```
$ sed -e '1!G;h;$!d' forward.txt > backward.txt
```

You'll find this sed script useful if you're logged in to a FreeBSD system, which doesn't happen to have a "tac" command. While handy, it's also a good idea to know why this script does what it does. Let's dissect it.

## Reversal explained

First, this script contains three separate sed commands, separated by semicolons: '1!G', 'h' and '$!d'. Now, it's time to get an good understanding of the addresses used for the first and third commands. If the first command were '1G', the 'G' command would be applied only to the first line. However, there is an additional '!' character -- this '!' character *negates* the address, meaning that the 'G' command will apply to *all but* the first line. For the '$!d' command, we have a similar situation. If the command were '$d', it would apply the 'd' command to only the last line in the file (the '$' address is a simple way of specifying the last line). However, with the '!', '$!d' will apply

the 'd' command to *all but* the last line. Now, all we need to to is understand what the commands themselves do.

When we execute our line reversal script on the text file above, the first command that gets executed is 'h'. This command tells sed to copy the contents of the pattern space (the buffer that holds the current line being worked on) to the hold space (a temporary buffer). Then, the 'd' command is executed, which deletes "foo" from the pattern space, so it doesn't get printed after all the commands are executed for this line.

Now, line two. After "bar" is read into the pattern space, the 'G' command is executed, which appends the contents of the hold space ("foo\n") to the pattern space ("bar\n"), resulting in "bar\n \foo\n" in our pattern space. The 'h' command puts this back in the hold space for safekeeping, and 'd' deletes the line from the pattern space so that it isn't printed.

For the last "oni" line, the same steps are repeated, except that the contents of the pattern space aren't deleted (due to the '$!' before the 'd'), and the contents of the pattern space (three lines) are printed to stdout.

Now, it's time to do some powerful data conversion with sed.

## sed QIF magic

For the last few weeks, I've been thinking about purchasing a copy of Quicken to balance my bank accounts. Quicken is a very nice financial program, and would certainly perform the job with flying colors. But, after thinking about it, I decided that I could easily write some software that would balance my checkbook. After all, I reasoned, I'm a software developer!

I developed a nice little checkbook balancing program (using awk) that calculates by balance by parsing a text file containing all my transactions. After a bit of tweaking, I improved it so that I could keep track of different credit and debit categories, just like Quicken can. But, there was one more feature I wanted to add. I recently switched my accounts to a bank that has an online Web account interface. One day, I noticed that my bank's Web site allowed me to to download my account information in Quicken's .QIF format. In very little time, I decided that it would be really neat if I could convert this information into text format.

## A tale of two formats

Before we look at the QIF format, here's what my checkbook.txt format looks like:

```
28 Aug 2000     food    -       -       Y     Supermarket             30.94

25 Aug 2000     watr    -       103     Y     Check 103               52.86
```

In my file, all fields are separated by one or more tabs, with one transaction per line. After the date, the next field lists the type of expense (or "-" if this is an income item). The third field lists the type of income (or "-" if this is an expense item). Then, there's a check number field (again, "-" if empty), a transaction cleared field ("Y" or "N"), a comment and a dollar amount. Now, we're ready to take a look at the QIF format. When I viewed my downloaded QIF file in a text viewer, this is what I saw:

```
!Type:Bank

D08/28/2000

T-8.15

N

PCHECKCARD SUPERMARKET

^

D08/28/2000

T-8.25

N

PCHECKCARD PUNJAB RESTAURANT

^

D08/28/2000

T-17.17

N

PCHECKCARD SUPERMARKET
```

After scanning the file, wasn't very hard to figure out the format -- ignoring the first line, the format is as follows:

```
D<date>

T<transaction amount>

N<check number>

P<description>

^

 (this is the field separator)
```

## Starting the process

When you're tackling a significant sed project like this, don't get discouraged -- sed allows you to gradually massage the data into its final form. As you progress, you can continue to refine your sed script until your output appears exactly as intended. You don't need to get it exactly right on the first try.

To start off, I created a file called "qiftrans.sed", and started massaging the data:

```
1d

/^^/d

s/[[:cntrl:]]//g
```

The first '1d' command deletes the first line, and the second command removes those pesky '^' characters from the output. The last line removes any control characters that may exist in the file. Since I'm dealing with a foreign file format, I want to eliminate the risk of encountering any control characters along the way. So far, so good. Now, it's time to add some processing punch to this basic script:

```
1d

/^^/d

s/[[:cntrl:]]//g

/^D/ {

 s/^D\(.*\)/\1\tOUTY\tINNY\t/

        s/^01/Jan/

        s/^02/Feb/

        s/^03/Mar/

        s/^04/Apr/

        s/^05/May/

        s/^06/Jun/

        s/^07/Jul/

        s/^08/Aug/

        s/^09/Sep/

        s/^10/Oct/

        s/^11/Nov/

        s/^12/Dec/

        s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:

}
```

First, I add a '/^D/' address so that sed will only begin processing when it encounters the first character of the QIF date field, 'D'. All of the commands in the curly braces will execute in order as soon as sed reads such a line into its pattern space.

The first line in the curly braces will transform a line that looks like:

```
D08/28/2000
```

into one that looks like thist:

```
08/28/2000 OUTY INNY
```

Of course, this format isn't perfect right now, but that's OK. We'll gradually refine the contents of the pattern space as we go. The next 12 lines have the net effect of transforming the date to a

three-letter format, with the last line removing the three slashes from the date. We end up with this line:

```
Aug 28 2000 OUTY INNY
```

The OUTY and INNY fields are serving as placeholders and will get replaced later. I can't specify them just yet, because if the dollar amount is negative, I'll want to set OUTY and INNY to "misc" and "-", but if the dollar amount is positive, I'll want to change them to "-" and "inco" respectively. Since the dollar amount hasn't been read yet, I need to use placeholders for the time being.

## Refinement

Now, it's time for some further refinement:

```
1d

/^^/d

s/[[:cntrl:]]//g

/^D/ {

        s/^D\(.*\)/\1\tOUTY\tINNY\t/

        s/^01/Jan/

        s/^02/Feb/

        s/^03/Mar/

        s/^04/Apr/

        s/^05/May/

        s/^06/Jun/

        s/^07/Jul/

        s/^08/Aug/

        s/^09/Sep/

        s/^10/Oct/

        s/^11/Nov/

        s/^12/Dec/

        s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:

        N

        N

        N

        s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/

        s/NUMNUM/-/

        s/NUM\([0-9]*\)NUM/\1/
```

```
        s/\([0-9]\),/\1/

}
```

The next seven lines are a bit complicated, so we'll cover them in detail. First, we have three 'N' commands in a row. The 'N' command tells sed to read in the *next line in the input* and append it to our current pattern space. The three 'N' commands cause the next three lines to be appended to our current pattern space buffer, and now our line looks like this:

```
28 Aug 2000 OUTY INNY \nT-8.15\nN\nPCHECKCARD SUPERMARKET
```

Sed's pattern space got ugly -- we need to remove the extra newlines and perform some additional formatting. To do this, we'll use the substitution command. The pattern we want to match is:

```
'\nT.*\nN.*\nP.*'
```

This will match a newline, followed by a 'T', followed by zero or more characters, followed by a newline, followed by an 'N', followed by any number of characters and a newline, followed by a 'P', followed by any number of characters. Phew! This regexp will match the entire contents of the three lines we just appended to the pattern space. But we want to reformat this region, not replace it entirely. The dollar amount, check number (if any) and description need to reappear in our replacement string. To do this, we surround those "interesting parts" with backslashed parentheses, so that we can refer to them in our replacement string (using '\1', '\2\, and '\3' to tell sed where to insert them). Here is the final command:

```
s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
```

This command transforms our line into:

```
28 Aug 2000  OUTY  INNY  NUMNUM    Y    CHECKCARD SUPERMARKET  AMT-8.15AMT
```

While this line is getting better, there are a few things that at first glance appear a bit...er...interesting. The first is that silly "NUMNUM" string -- what purpose does that serve? You'll find out as you inspect the next two lines of the sed script, which will replace "NUMNUM" with a "-", while "NUM"<number>"NUM" will be replaced with <number>. As you can see, surrounding the check number with a silly tag allows us to conveniently insert a "-" if the field is empty.

## Finishing touches

The last line removes a comma following a number. This converts dollar amounts like "3,231.00" to "3231.00", which is the format I use. Now, it's time to take a look at the final, production script:

```
1d

/^^/d

s/[[:cntrl:]]//g

/^D/ {

 s/^D\(.*\)/\1\tOUTY\tINNY\t/
```

```
 s/^01/Jan/
 s/^02/Feb/
 s/^03/Mar/
 s/^04/Apr/
 s/^05/May/
 s/^06/Jun/
 s/^07/Jul/
 s/^08/Aug/
 s/^09/Sep/
 s/^10/Oct/
 s/^11/Nov/
 s/^12/Dec/
 s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:
 N
 N
 N
 s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
 s/NUMNUM/-/
 s/NUM\([0-9]*\)NUM/\1/
 s/\([0-9]\),/\1/
 /AMT-[0-9]*.[0-9]*AMT/b fixnegs
 s/AMT\(.*\)AMT/\1/
 s/OUTY/-/
 s/INNY/inco/
 b done
:fixnegs
 s/AMT-\(.*\)AMT/\1/
 s/OUTY/misc/
 s/INNY/-/
:done
}
```

The additional eleven lines use substitution and some branching functionality to perfect the output. We'll want to take a look at this line first:

```
        /AMT-[0-9]*.[0-9]*AMT/b fixnegs
```

This line contains a branch command, which is of the format "/regexp/b label". If the pattern space matches the regexp, sed will branch to the fixnegs label. You should be able to easily spot this label, which appears as ":fixnegs" in the code. If the regexp doesn't match, processing continues as normal with the next command.

Now that you understand the workings of the command itself, let's take a look at the branches. If you look at the branch regular expression, you'll see that it will match the string 'AMT', followed by a '-', followed by any number of digits, a '.', any number of digits and 'AMT'. As I'm sure you've figured out, this regexp deals specifically with a negative dollar amount. Earlier, we surrounded our dollar amount with 'AMT' strings so we could easily find it later. Because the regexp only matches dollar amounts that begin with a '-', our branch will only happen if we happen to be dealing with a debit. If we are dealing with a debit, OUTY should be set to 'misc', INNY should be set to '-', and the negative sign in front of the debit amount should be removed. If you follow the code, you'll see that this is exactly what happens. If the branch isn't executed, OUTY gets replaced with '-', and INNY gets replaced with 'inco'. We're finished! Our output line is now perfect:

```
28 Aug 2000 misc - -       Y     CHECKCARD SUPERMARKET  -8.15
```

## Don't get confuSed

As you can see, converting data using sed isn't all that hard, as long as you approach the problem incrementally. Don't try to do everything with a single sed command, or all at once. Instead, gradually work your way toward the goal, and continue to enhance your sed script until your output looks just the way you want it to. Sed packs a lot of punch, and I hope that you've become very familiar with its inner workings and that you'll continue to grow in your sed mastery!

## Related topics

- Read Daniel's previous sed articles on *developerWorks*: Common threads: Sed by example, Part 1 and Part 2.
- If you'd like a good old-fashioned book, O'Reilly's sed & awk, 2nd Edition would be wonderful choice.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.